Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Módulos software para un sistema de asistencia a la movilidad basado en entorno ROS (Robot Operating System)

Autor: Francisco Javier La Roda Herrero

Tutor: Juan Carlos García García

2018

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Módulos software para un sistema de asistencia a la movilidad basado en entorno ROS (Robot Operating System)

Autor: Francisco Javier La Roda Herrero

Director: Juan Carlos García García

Tribunal:

Presidente: Álvaro Hernandez Alonso

Vocal 1º: Pablo Ramos Sainz

Vocal 2º: Juan Carlos García

Calificación: .....

Fecha: .....

"La frase más excitante que se puede oír en ciencia, la que anuncia nuevos descubrimientos, no es "¡Eureka!" sino Es extraño..."

Asimov

Agradecimientos

Este Trabajo de Fin de Grado es la culminación del sueño que tengo desde pequeño de convertirme en Ingeniero y que estuve a punto de abandonar después de unos primeros años difíciles.

He puesto en este TFG todo el conocimiento que he adquirido tanto en la universidad como fuera de ella y muchas horas de investigación, trabajo y también frustración.

En primer lugar quiero agradecer a mis compañeros y ya amigos, que me han acompañado durante estos años de carrera, siendo uno de los mejores periodos de mi vida. Especialmente quería mencionar a Alex, Álvaro, Daoud, Edu, Ernesto, Fran, Juanjo, Oscar, Roni y Simo.

También quiero agradecer a mi familia, que me han aguantado tantos años y especialmente a mis padres, que han puesto todo lo que tienen para ayudarme a superar este desafío.

A mis amigos de siempre y de ahora.

Para terminar quiero agradecer a mis tutores, Marta y Juan Carlos, por confiar en mí para hacer este TFG y haberme ayudado a lo largo de este año y medio de trabajo en el me han enseñado muchas cosas.

Resumen

El *software* libre ha permitido el acceso a la comunidad, a *software* que normalmente solo podía ser accesible por las empresas. ROS (*Robotic Operating System*), es un sistema operativo que permite crear complejos sistemas robóticos con un bajo presupuesto y con *hardware* comercial.

Continuando los trabajos del Departamento de Electrónica sobre una silla de ruedas robotizada, se ha implementado un sistema de navegación autónomo basado en la odometría y la detección de obstáculos usando la tecnología desarrollada bajo ROS.

Palabras clave: ROS, Silla de Ruedas, Navegación Autónoma, Robot Móvil, Control.

Abstract

Free software brings to the community software that usually was affordable only by the industry. ROS (Robotic Operating System), is an Operative System that allows to create complex robotic systems with low budget and commercial hardware.

Continuing the works of the Department of Electronics on a robotic wheelchair, an autonomous navigation system has been implemented, based on odometry and obstacle detection, using the technology developed by ROS.

Keywords: ROS, Wheelchair, Autonomous Navigation, Mobile Robot, Control.

Resumen extendido

En este Trabajo Fin de Grado, se ha implementado un sistema de navegación autónoma basado en el sistema operativo ROS, sobre una silla de ruedas (SARA) desarrollada por el Departartamento de Eletrónica durante los últimos años.

Se ha tomado como punto de partida el trabajo desarrollado por Javier León [1], en que se ha implementado la comunicación entre el bajo nivel (motores, encoder, sensores) y ROS, instalado en un ordenador.

La línea de trabajo seguida para implementar el sistema de navegación es la siguiente.

- Primero se ha modificado el *script lectura.py* del trabajo de Javier León para hacer compatibles los *topics* con el resto de paquetes de ROS.
- A continuación ha sido necesario implementar un sistema de control de alto nivel mediante el paquete *ros_control*. El controlador de alto nivel permite realizar el control de velocidad lineal (avance) y velocidad angular (giro), mediante la realimentación proporcionada por los encoders.
- Una vez desarrollado el control, se ha implementado la navegación gracias al paquete de *ros_navigation*.
- Para finalizar se han realizado un conjunto de pruebas que verifican el funcionamiento del sistema.

A continuación se explican de forma resumida los principales apartados de este trabajo.

Bajo nivel

La silla de ruedas cuenta con varios módulos *hardware* y *sofware* diseñados en trabajos previos. Estos módulos se comunican con el alto nivel mediante un bus CAN, que es encapsulado en el protocolo USB, para permitir la comunicación con un PC.

En el PC se ha instalado ROS, y con el paquete *canusb* implementado en anteriores trabajos se ha podido desencapsular las tramas CAN y convertirlas en los mensaje que usa ROS, los *topic*. Ver imagen 2.

- Los *topic canrx* y *cantx* permiten la recepción y transmisión de tramas can en crudo. La estructura del mensaje se describe en apartado 3.2.5.
- La trama CAN recibida en el *topic canrx* se demultiplexa en función del código del nodo recibido, dando lugar al resto de *topics* descritos en la imagen 2.

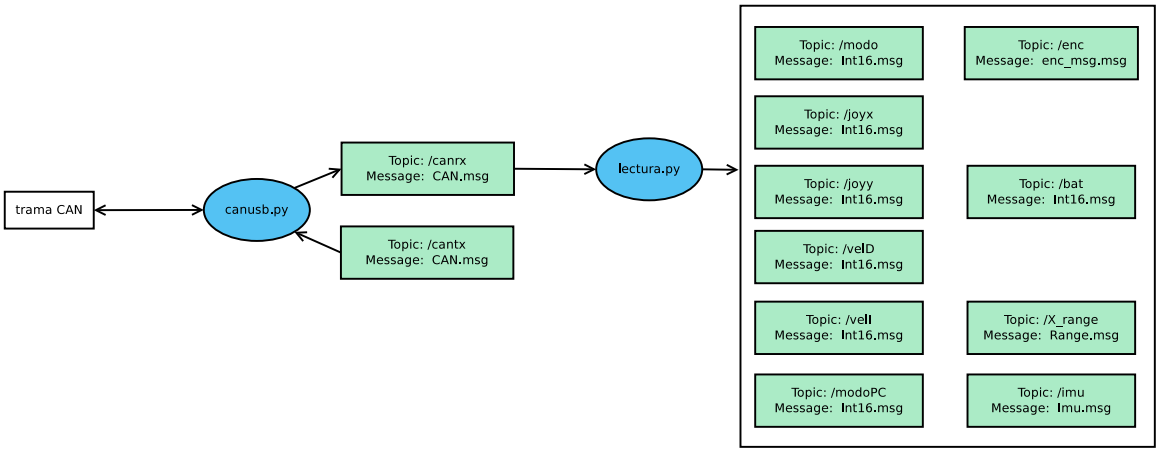


Figura 2: Topics creados por el paquete canusb

En este trabajo se ha modificado el fichero *lectura.py* donde se demultiplexan los *topic*. En el trabajo de partida se usaban mensajes genéricos de tipo *int*, pero ha sido necesario modificarlos para hacerlos compatibles con el resto de paquetes.

Por ejemplo se ha usado el mensaje *Range*, específico para sensores de distancia o el mensaje *Imu* para la unidad inercial.

También se ha creado un mensaje personalizado para transmitir la información de los encoder, que junte en el mismo mensaje la información de los pulsos y la marca temporal del bajo nivel.

Control de Alto nivel

El control de alto nivel se ha implementado mediante *ros_control*, un conjunto de paquetes de ROS que permiten usar una gran cantidad de controladores desarrollados por la comunidad. En este trabajo se ha usado el controlador *diff_drive_controller*, diseñado específicamente para robots con tracción diferencial 2.2, como la silla de ruedas.

Los controladores de *ros_control* son completamente genéricos por lo que se pueden adaptar al *hardware* de cualquier robot. Para ello es necesario crear una *interface hardware*, que permite la comunicación entre el controlador y el hardware de la silla. Este código se ha programado en C++ ya que hace uso de una serie de librerías de *ros_control* que solo están disponibles en este lenguaje.

En la imagen 3 se muestra un esquema con los distintos nodos del controlador y como se comunican con el paquete *canusb*.

En la imagen 4 se muestra con más detalle el funcionamiento de la interface hardware.

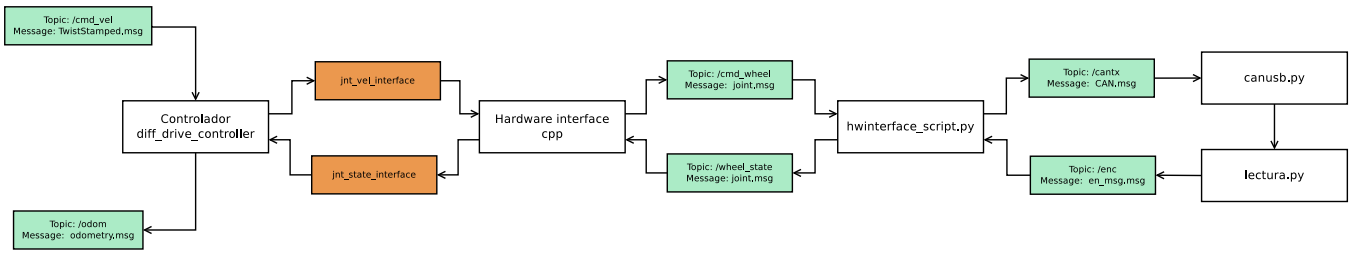


Figura 3: Esquema general del control de alto nivel.

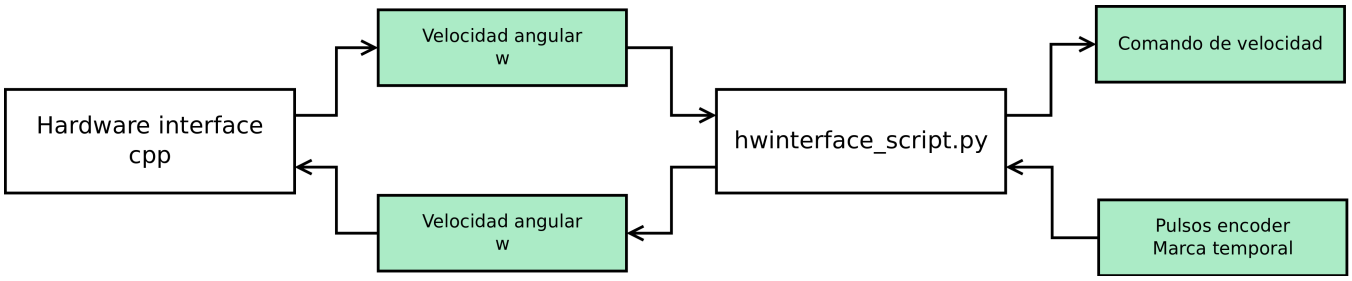


Figura 4: Esquema de la interface entre hardware_interface y la silla.

Sistema de navegación (*ros_navigation*)

El sistema de navegación se ha implementado mediante *ros_navigation*, un conjunto de paquetes de ROS que permiten desarrollar un sistema de navegación usando diversos algoritmos ya desarrollados.

El núcleo de *ros_navigation* es el *Navigation Stack*, que cuenta con todo lo necesario para hacer posible la navegación [2]. Existen además multitud de paquetes complementarios para expandir el funcionamiento de *ros_navigation*. En la figura 5 se muestra un esquema general de los componentes de *Navigation Stack*.

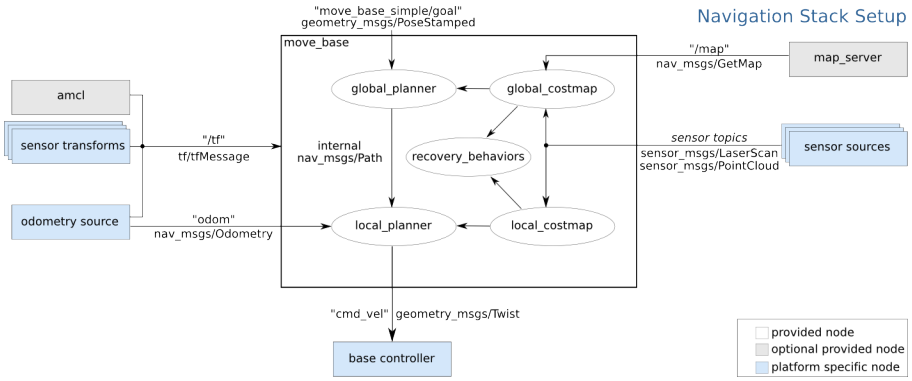


Figura 5: *Navigation Stack*

Navigation Stack cuenta con los siguientes bloques:

- *move_base* (recuadrado en la figura 5): Realiza el trabajo principal de navegación.
- *map_server*: Proporciona el mapa sobre el que se va a realizar la navegación.
- *odometry source*: Información de la odometría. En este trabajo la información de odometría es generada por el controlador de alto nivel a partir de los pulsos de los encoders *encoders*.
- *sensor sources*: Información proveniente de los sensores. Es posible usar numerosos tipos de sensores, pero ROS Navigation está centrado en el uso de sensores láser.
- *amcl (adaptative Mote Carlo localization)*: Sistema probabilístico que hace uso de los sensores láser para determinar la ubicación absoluta de un robot. En este trabajo no se ha usado.
- *sensor transform*: *Offset* de posición entre los sensores y el entorno, lo que permite calcular la posición del robot en el espacio. Ver sección 4.3 para conocer más a fondo el funcionamiento de las transformaciones de ROS.
- *base controller*: Representa el robot, al que se envían los comandos de velocidad.

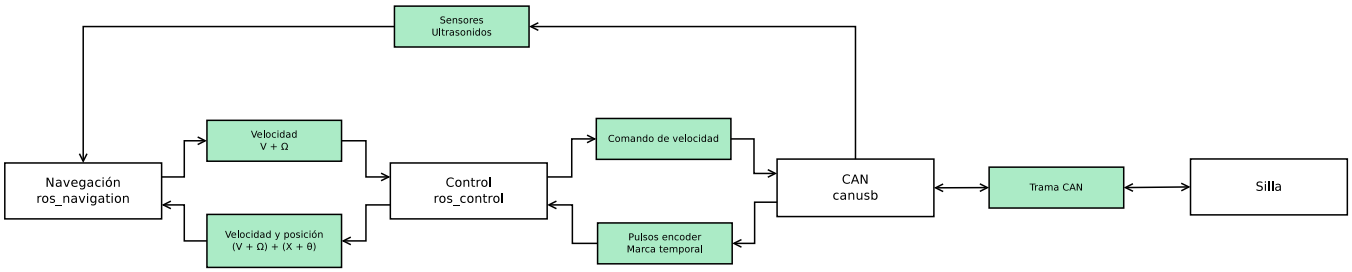


Figura 6: Esquema general del sistema.

Una vez se han configurado correctamente todos los paquetes, es necesario conectar el sistema de navegación con el resto del proyecto. La estructura general del sistema, se muestra en la imagen 6.

Interface de usuario (rviz)

En este momento el sistema de navegación ya está implementado, pero es necesario una interface que permita interactuar con los usuarios y donde poder introducir el destino al que se quiere llegar.

Para ello se ha usado el paquete *rviz*, que está diseñado para leer y mostrar adecuadamente los *topics* de un sistema de navegación 2d.

En la imagen 7 se muestra la *interface* que crea *rviz*, con la que se controla la silla.

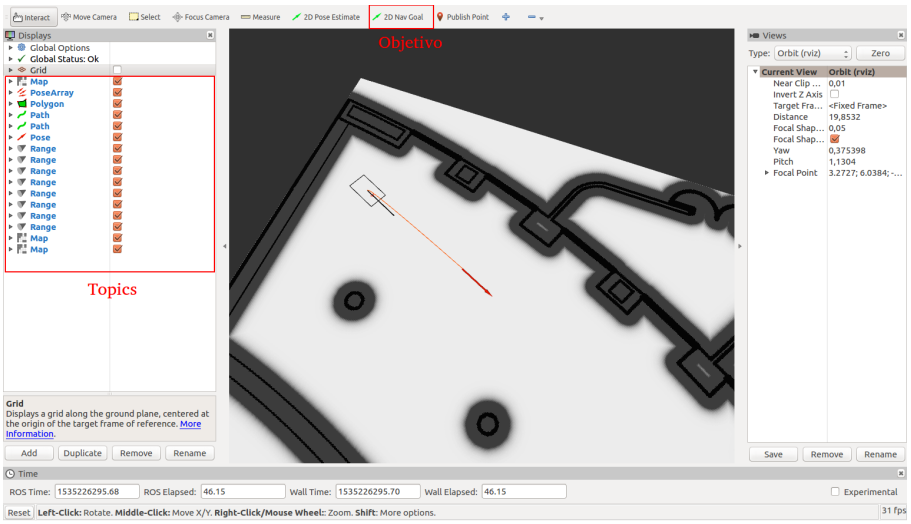


Figura 7: Visualizador Rviz.

En este vídeo se muestra un ejemplo de funcionamiento del sistema de navegación con rviz, donde se han introducido varias metas y el robot se ha desplazado hasta alcanzarlas: https://www.youtube.com/watch?v=Ll5xFwT535o.

Resultados

Se han realizado pruebas de navegación para determinar el comportamiento de la silla con el sistema de navegación implementado.

Las metas en el mapa se han generado mediante el paquete de ROS SimpleActionClient [3], que permite generar un recorrido personalizado introduciendo una secuencia de metas.

En las imágenes 8, 9, 10 y 11 se muestra una de las pruebas realizadas en la terraza. Se puede apreciar como el sistema de navegación crea la ruta necesaria para evadir la columna.

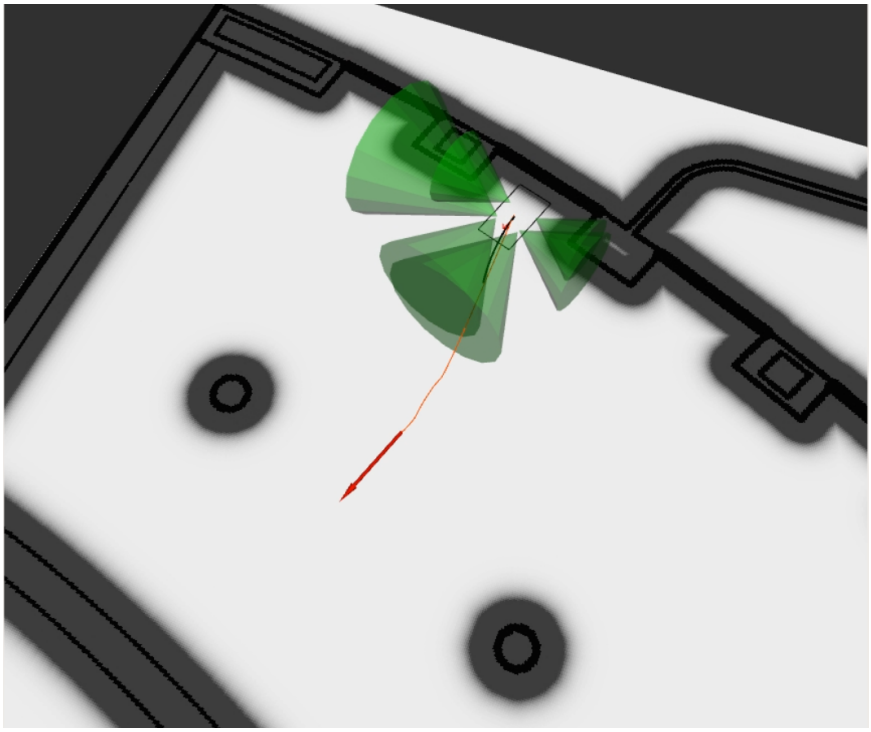


Figura 8: Recorrido realizado en el mapa de la terraza, de forma estática.

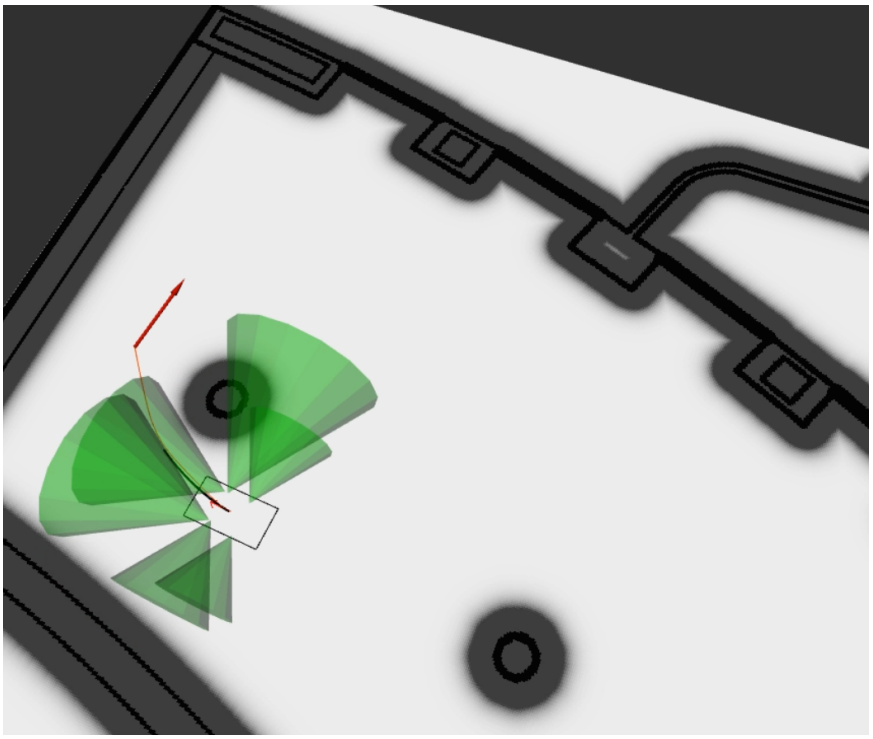


Figura 9: Recorrido realizado en el mapa de la terraza, de forma estática.

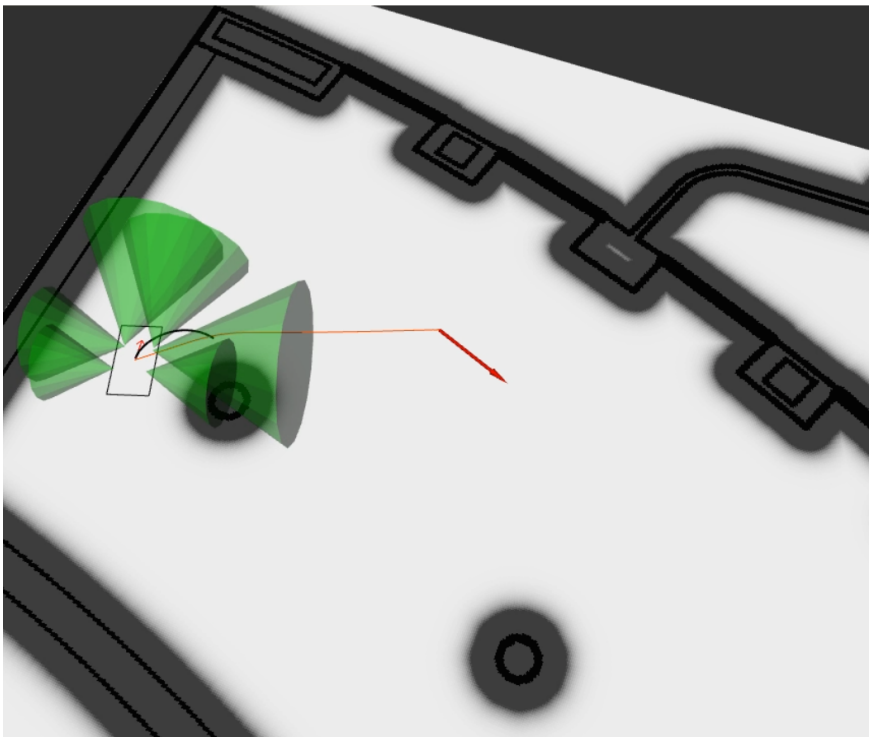


Figura 10: Recorrido realizado en el mapa de la terraza, de forma estática.

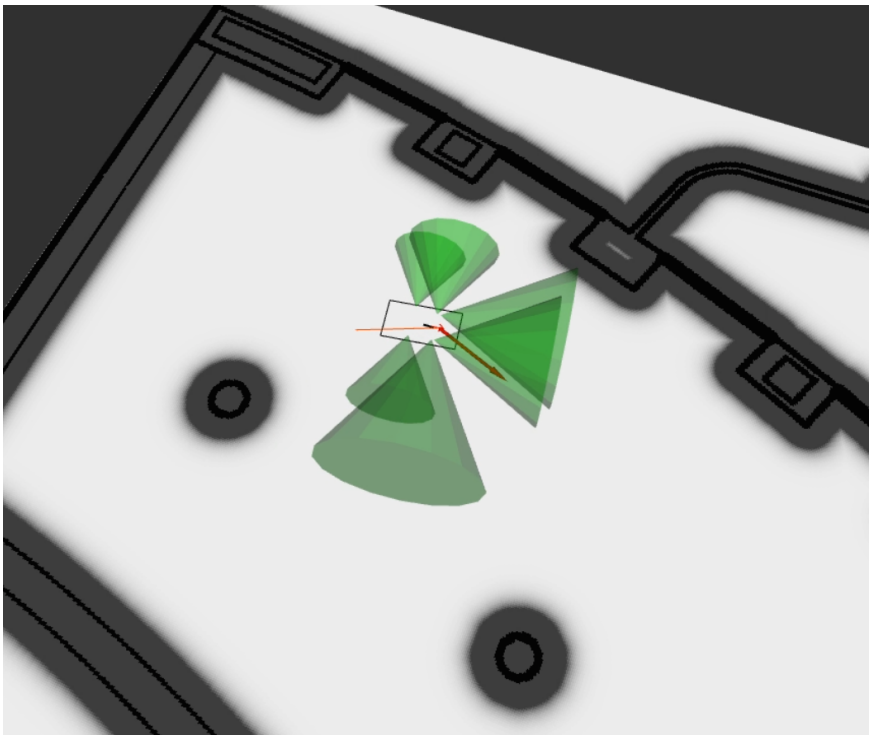


Figura 11: Recorrido realizado en el mapa de la terraza, de forma estática.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este Trabajo Fin de Grado continúa el proyecto "Diseño de módulos software para un sistema avanzado robótico de asistencia a la movilidad basado en entorno de desarrollo robótico ROS" de Javier León Almazán [1]. A su vez el trabajo de Javier León es una continuación de una larga lista de trabajos y proyectos [4], [7], [8], [9], [6], [10], [11], que tienen como objetivo principal el dotar a la silla de ruedas de un sistema de navegación autónomo y de distintos métodos de entrada para controlarla manualmente.

Con la llegada de ROS en el año 2007, se migró a esta plataforma, que permite la integración de algoritmos de navegación y control de forma sencilla y modular, con un gran soporte de la comunidad. David Pinedo [11], inició la migración de los algoritmos de alto nivel al *framework* de ROS, manteniendo los sistemas de bajo nivel intactos.

Todo el sistema de alto nivel se ejecutará desde un portátil dotado de pantalla táctil, que permitirá elegir las metas sobre un mapa de forma sencilla. Puesto que el *hardware* usado es comercial, es posible actualizar el sistema de forma sencilla en un futuro, en caso de necesitar más recursos computacionales. Además facilita la incorporación de sistemas adicionales como sensores y cámaras.

1.1 Objetivos

El objetivo de este trabajo es dotar a la silla de un sistema de navegación reactiva, con ayuda de la odometría y los sensores de ultrasonidos instalados. Para ello se han usado múltiples paquetes de ROS, que se describen en los siguientes apartados de esta memoria.

Como base se ha tomado el trabajo en materia de *software* desarrollado por Javier León [1] y David Pinedo [11], que implementaron la comunicación entre el bajo nivel y ROS mediante el bus CAN.

Además se apoya en el *hardware* desarrollado por el resto de proyectos referenciados en el apartado anterior.

1.2 Estructura de la memoria

La memoria está distribuida en 7 capítulos y un anexo, que explican el proceso de análisis y desarrollo de este trabajo.

- En el Capítulo 2, se realiza una introducción de la teoría necesaria para entender y realizar este trabajo.

- En el capítulo 3, se analiza el trabajo realizado por los anteriores compañeros y se explican los cambios realizados en el *software* que han sido necesario realizar para desarrollar este proyecto.
- En el capítulo 4, se expone la implementación del sistema de control y el sistema de navegación realizada en este TFG, mediante el *framework* de ROS.
- En el capítulo 5, se explican los experimentos realizados y los resultados obtenidos.
- En el capítulo 6 se muestran las conclusiones obtenidas al finalizar el trabajo y los posibles trabajos futuros.
- En los capítulos 7 y 8 se detallan los recursos utilizados y el coste del proyecto.
- En el anexo A se proporciona un manual para instalar y poner el marcha el *software* desarrollado en este proyecto.

Capítulo 2

Base teórica

2.1 Introducción

En este capítulo se describe la cinemática de la silla, que permite desarrollar el algoritmo para extraer la posición de la silla a partir de los encorder.

El algoritmo está dividido en varias bloques como se muestra en la imagen 2.1. Estos bloques son descritos en los siguientes apartados.

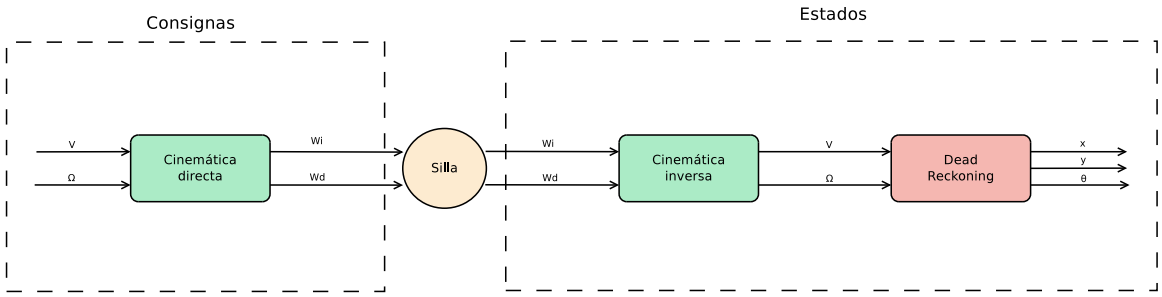


Figura 2.1: Diagrama explicativo de la cinemática, cinemática inversa y *dead reckoning* en la silla

2.2 Cinemática directa

La cinemática describe el comportamiento dinámico del robot ante una entrada de velocidad lineal (V) y velocidad de rotación (Ω).

La silla de ruedas tiene un modelo cinemático de tracción diferencial, que se caracteriza por tener dos ruedas motrices separadas una cierta distancia, como se muestra en la imagen 2.2.

Las ecuaciones 2.1 y 2.2 describen el modelo cinemático a partir de los parámetros físicos de la silla que se detallan a continuación.

$$\omega_D = \frac{1}{R} * (V + \Omega * \frac{D}{2})[rad/s] \tag{2.1}$$

$$\omega_I = \frac{1}{R} * (V - \Omega * \frac{D}{2})[rad/s] \tag{2.2}$$

- ω_D ,ω_I: Velocidad angular de las ruedas derecha e izquierda. $[rad/s]$

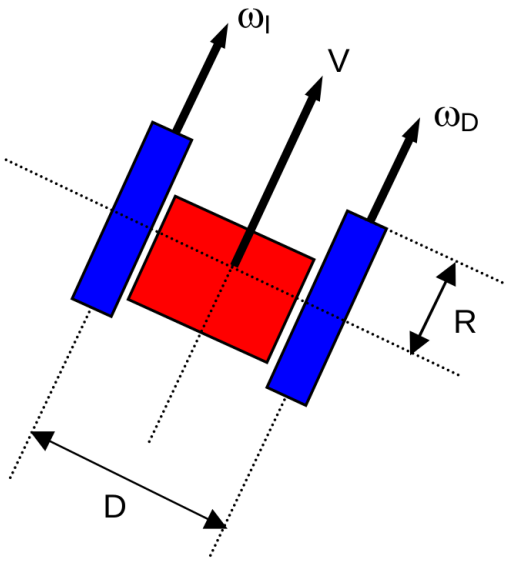


Figura 2.2: Diagrama explicativo de la cinemática directa de la silla de ruedas [4].

- D: Separación de las ruedas. $[m]$
- R: Radio de las ruedas. $[m]$
- V: Velocidad lineal de la silla. $[m/s]$
- Ω: Velocidad de rotación de la silla. $[rad/s]$

2.3 Odometría: cinemática inversa y *dead reckoning*

La odometría permite conocer la posición relativa al punto de partida usando la velocidad angular de las ruedas ω, proporcionada por los encoders.

Para ello se aplican las ecuaciones de cinemática inversa y *dead reckoning* que se describen a continuación.

- Cinemática inversa.

$$V = \frac{R}{2} * (\omega_D + \omega_I) \tag{2.3}$$

$$\Omega = \frac{R}{D} * (\omega_D - \omega_I) \tag{2.4}$$

- *Dead reckoning* discreto con muestreo T_s

$$x_k = x_{k-1} + T_s * V_k cos(\theta)[m] \tag{2.5}$$

$$y_k = y_{k-1} + T_s * V_k sen(\theta)[m] \tag{2.6}$$

$$\theta_{0k} = \theta_{0k-1} + T_s * \Omega_k[rad] \tag{2.7}$$

El proceso de *dead reckoning* mostrado en la imagen 2.4 permite obtener el desplazamiento de la silla de ruedas cada kTs, conociendo la posición anterior de la misma $(x_{k-1}, y_{k-1}, \theta_{k-1})$ y mediante el modelo discretizado, tal y como se muestra en las ecuaciones 2.5, 2.6, 2.7.

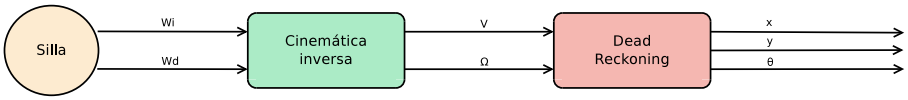


Figura 2.3: Diagrama explicativo de la cinemática inversa y *deadreckoning*

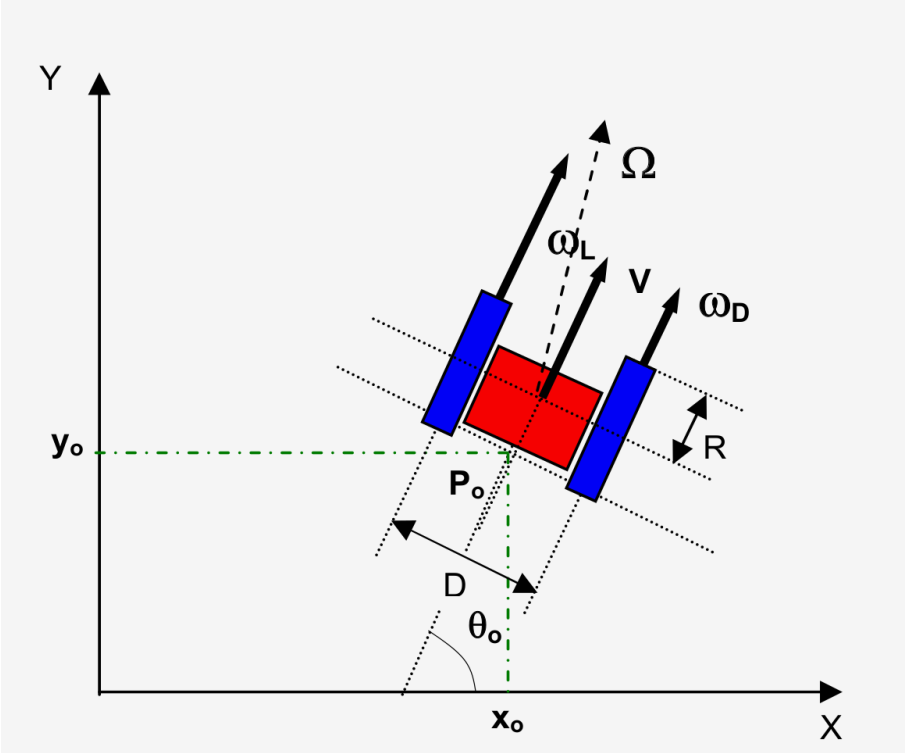


Figura 2.4: Diagrama explicativo de la cinemática inversa del robot diferencial [4].

Además, queda en este punto de manifiesto el problema que presenta el uso de un posicionador relativo como el de *dead reckoning* en estos casos. Debido a su constitución, la silla de ruedas se comporta como un sistema no holonómico, pues el estudio independiente de cada una de las partes móviles no basta para obtener el movimiento absoluto del sistema. Es por ello que para conocer la posición real del móvil es necesario integrar, con lo que los errores de posicionamiento cometidos en el proceso de integración puede convertirse en errores de posición graves, pues son acumulativos.

Más información en [7].

2.4 Aplicación en ROS

El objetivo de usar ROS en este TFG y en trabajos previos, es utilizar en la medida que sea posible los paquetes disponibles creados por la comunidad de desarrolladores de ROS. Esto permite ahorrar tiempo en el desarrollo y centrarse en elegir los paquetes más idóneos para una aplicación concreta. Además estos paquetes son mejorados continuamente por la comunidad de desarrolladores y por lo tanto los proyectos implementados no se quedan obsoletos.

Por otro lado esta metodología de trabajo asegura la modularidad del proyecto resultante lo que permite reutilizar partes del trabajo para otros proyectos futuros e integrar fácilmente nuevas funcionalidades.

En este trabajo se ha usado el paquete *ros_control*, junto con el controlador *diff_drive_controller*

para realizar los procesos de cinemática directa, cinemática inversa y *dead reckonig*, sin necesidad de implementar ningún algoritmo.

Además este controlador está especializado para su uso de robots diferenciales, por lo que no ha sido necesario modelar la planta.

Este paquete se explica con detalle en la sección 4.1.3.

Capítulo 3

Análisis y modificación del sistema de robotizado de partida

3.1 Introducción

En este capítulo se va a describir la estructura del sistema de robotizado de partida.

- En primer lugar se va a describir brevemente la estructura del *hardware* y *software* de bajo nivel.
- Posteriormente se describe la interface realizada entre el bajo nivel y ROS.
- Para terminar se muestra el modelo de la silla de ruedas creado para ROS.

El diagrama 3.1 representa los bloques principales que componen el sistema.

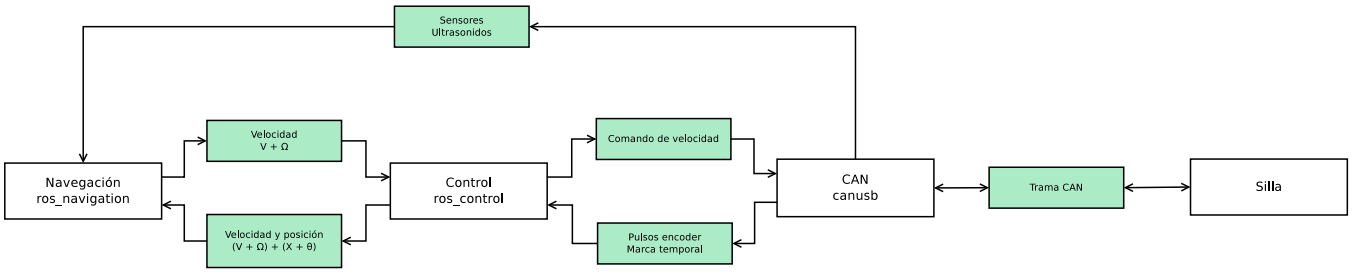


Figura 3.1: Esquema general del sistema.

3.2 *Hardware* y *software* de bajo nivel

3.2.1 Introducción

Es este apartado se describe la estructura *hardware* de la silla de ruedas robotizada, que cuenta con varios nodos que se describen en los siguientes apartados.

Los nodos se comunican entre ellos mediante bus CAN, como se describe en el esquema de la imagen 3.2.

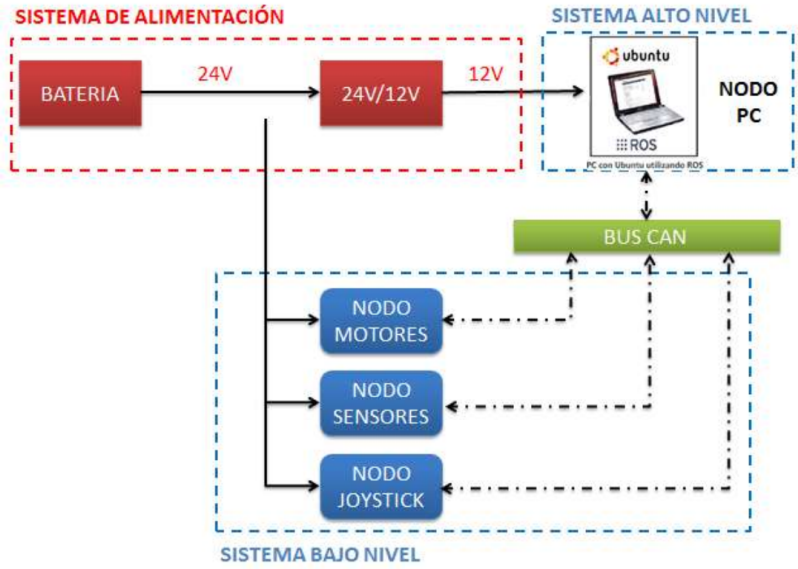


Figura 3.2: Nodos *hardware*.

3.2.2 Nodo Motores

El nodo motores está constituido por 2 motores de imanes permanentes con sus correspondientes drivers AX3500 y enconders ópticos HEDS-5500A11 7.1 encastrados en ambas ruedas.

El driver de los motores AX3500 cuenta con un controlador PID integrado, que se activa para realizar un control de bajo invel de la velocidad angular de cada una de las ruedas de la silla. En el apartado 4.1.2 se precisa su funcionamiento.

Los enconders cuentan con dos discos de 500 huecos cada uno, por lo que cada vuelta se traduce en 2000 flancos. Los motores cuentan con una reductora de valor 32, por lo que cada vuelta de la rueda se traduce en 64000 cuentas.

3.2.3 Nodo Sensores

El nodo sensores lo forman nueve sensores de ultrasonidos SRF02, que incluyen el emisor, receptor y sus correspondientes etapas adaptadoras de tensión.

La información de los sensores es concentrada en la tarjeta LPC2129, con la cual se comunican mediante un bus serie con protocolo I2C. Esta tarjeta encapsula los datos en una trama CAN para ser enviada al PC

Además, también está incorporado en este nodo (en el mismo bus I2C) un acelerómetro para la detección de choques en la silla, aunque su gestión no está implementada.

La disposición física de los sensores se muestra en la imagen 3.3, los cuales hay que añadir el sensor ubicado bajo el *joystick* que no se muestra en la imagen.

3.2.4 Nodo Joystick

El nodo joystick incluye una interfaz hombre-máquina compuesta de los siguientes elementos: display, teclado, bocina, luz de encendido, joystick analógico de dos ejes y potenciómetro para seleccionar la velocidad.

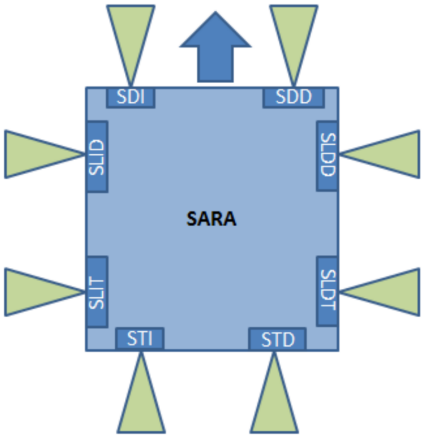


Figura 3.3: Ubicación de los sensores en la silla.

Para conocer más información sobre las características de estos elementos consultar el documento [10].

Con la interfaz hombre-máquina mencionada, al poner en funcionamiento la silla se pueden elegir entre 4 modos de funcionamiento distintos:

- **Modo joystick.** En este modo la silla de ruedas se mueve manualmente mediante un joystick analógico. Corresponde al modo 1 en la silla de ruedas.
- **Modo soplido.** En este modo la silla se mueve usando un detector de soplo a modo de joystick digital (ver [8]). Corresponde al modo 2 en la silla de ruedas.
- **Modo PC.** Este modo es el usado en este TFG para conectar el bajo nivel con la plataforma de navegación de ROS. Corresponde al modo 3 en la silla de ruedas.
- **Modo joystick digital.** En este modo la silla de ruedas se controla también manualmente a través del joystick, pero de manera digital, es decir, mediante toques interpretados por un *software* ad-hoc (ver [10]). Corresponde al modo 4 en la silla de ruedas.

3.2.5 Estructura de las tramas CAN

Las tramas de los mensajes CAN tienen la estructura mostrada en la imagen 3.4 cuyos campos se describen a continuación:

Figura 3.4: Trama CAN.

- **SOF.** Marca el inicio de la trama.
- **Arbitration Field.** Identificador de la trama. En la trama estándar tiene un tamaño de 12 bits y en la trama extendida de 32 bits.

- **Control field.** Campo de control. Los dos primeros bits están reservados y los 4 restantes indican el número de bytes de datos incluidos en la trama.
- **Data Field.** Campo de datos de la trama.
- **CRC.** Código de redundancia cíclica. Con este código se comprueba si hay o no errores en la trama.
- **ACK.** Celda de reconocimiento. Indica si la trama ha sido recibida correctamente.
- **EOF.** Marca el fin de la trama.

Como se observa en la Tabla 3.1, por el bus CAN implementado en la silla de ruedas se envían los datos recibidos en cada momento del bajo nivel que se listan a continuación:

- **Joystick** Cada 100ms se envía una trama.
  - **Joyx, Joyy.** Valores de los ejes de abscisas y ordenadas (eje x y eje y), respectivamente, del joystick.
  - **velI, velD.** Valores de velocidad de las ruedas izquierda y derecha respectivamente.
  - **modo.** Modo de funcionamiento de la silla (ver al principio de este apartado).
  - **modo PC.** Este mensaje a diferencia del resto, solo se envía una única vez al acceder al modo PC o al salir de este, en lugar de recibirse un envío periódico. Este mensaje es importante en el desarrollo del TFG ya que lo utiliza el bajo nivel para comprobar si existe comunicación a través del bus CAN con el PC. Funciona de la siguiente manera:
    - Si se accede al modo PC (pulsando el 3 en el teclado de la silla), se envía un solo mensaje con el valor 3 por la trama. Por seguridad, solo se permite el envío de consignas de velocidad angular a las ruedas desde el sistema de navegación ROS cuando se activa este modo.
    - Si se sale del modo PC (pulsando el 0 en el teclado de la silla), se envía un solo mensaje con el valor 0 por la trama.
- **Motores.** Cada 100ms se envía una trama.
  - **encoderAABS, encoderBABS.** Valores (en cuentas) de los encoders A y B, respectivamente. Cada vuelta de la rueda corresponde con 64000 cuentas.
  - **tAabs, tBabs.** Marca temporal desde la puesta en marcha, hasta la lectura actual de los encoders A y B, respectivamente. Tienen una resolución de 100us por cuenta. (Para más información ver [8] y [9]).
  - **nivel de batería.**. Voltaje instantáneo de la bateria. Resolución de 0.1V por cada cuenta.
- **Sensores.** Cada 100ms se envía una trama.
  - **SDI, STD SLDD, SLIT, SLID, SDD, SLDT, STI, SJI.** Datos de los nueve sensores de ultrasonidos que lleva la silla, siendo S Sensor, D Delantero o Derecho, I Izquierdo y T Trasero.
  - **EjeX, EjeY y EjeZ.** Lectura de los 3 ejes del acelerómetro. Rango de medida 3.6g Sensibilidad de 300 $\frac{mv}{g} = 93\ \frac{Dato}{g} = 9.49\ \frac{Dato}{m/s^2}$ (Para más información ver [8].)
- **PC**
  - **DatoD, DatoI y lazo.** Permite enviar las consignas de velocidad para excitar los dos motores y controlar el funcionamiento del control PID (lazo abierto o cerrado).

    Los comandos de la velocidad son los siguientes:

- (1-127): Velocidad positiva.
- (129-255): Velocidad negativa.
- (0 y 129): Motor Parado.

Para modificar el comportamiento del lazo se usan los siguientes caracteres en el mensaje de lazo:

- Carácter 'A': Lazo abierto.
- Carácter 'C': Lazo cerrado.

Tabla 3.1: Formato de las tramas del bus CAN en la silla de ruedas

EMISOR	IDENTIFICADOR	MENSAJE	TIPO	TAMAÑO
JOYSTICK	0x110 (272)	Joyx	short	2 bytes
		velI	char	1 byte
		velD	char	1 byte
		Joyy	short	2 bytes
		modo	short	2 bytes
	0x111 (273)	modo(para PC)	short	2 bytes
		—	VACIO	2 bytes
MOTORES	0X101 (257)	encoderAABS	int	4 bytes
		tAabs	int	4 bytes
	0x102 (258)	encoderBABS	int	4 bytes
		tBabs	int	4 bytes
	0x210 (528)	nivel batería	int	4 bytes
		—	VACIO	4 bytes
SENSORES	0x201 (513)	SDI	short	2 bytes
		STD	short	2 bytes
		SLDD	short	2 bytes
		SLIT	short	2 bytes
	0x202 (514)	SLID	short	2 bytes
		SDD	short	2 bytes
		SLDT	short	2 bytes
		STI	short	2 bytes
	0x203 (515)	EjeZ	short	2 bytes
		SJOYS	short	2 bytes
		EjeX	short	2 bytes
		EjeY	short	2 bytes
PC	0x120 (288)	DatoD	char	1 bytes
		DatoI	char	1 bytes
		—	VACIO	2 bytes
		Lazo	char	1 bytes
		—	VACIO	2 bytes

3.3 *Interface hardware-software (canusb)*

3.3.1 Introducción

Como se se ha expuesto, este sistema se compone de dos bloques principales, la parte de *hardware* o bajo nivel, descrita en el apartado anterior y la parte de alto nivel que se ejecuta en un PC.

Para poder transmitir los mensajes CAN que usan los nodos para comunicarse con el PC, estos se encapsulan en una comunicación USB. Para ello se usa una tarjeta de conversión USB-CAN 7.1.

Por lo tanto en el PC es necesario un *software* que permita desencapsular las tramas CAN y para ello se va a utilizar el paquete de ROS *canusb*.

3.3.2 Funcionamiento

El paquete de ros *canusb* diseñado en anteriores trabajos [11], [1], permite la comunicación mediante tramas CAN entre el bajo nivel y ROS.

La comunicación en ROS se basa en *topics*, por lo que este paquete se encarga de transformar las tramas CAN en la estructura de *topics* que se describe en la imagen3.5.

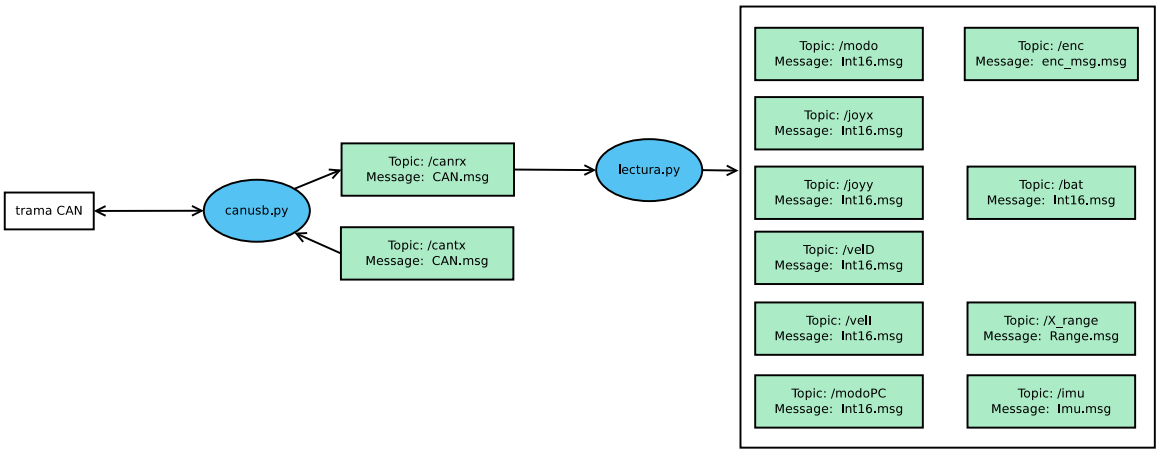


Figura 3.5: *Topics* creados por el paquete *canusb*

- Los *topic canrx* y *cantx* permiten el la recepción y transmisión de tramas CAN en crudo. La estructura del mensaje se describe en apartado 3.2.5.
- La trama CAN recibida en el *topic canrx* se demultiplexa en función del nodo del que provenga la información, dando lugar al resto de *topics* descritos en la figura.

3.3.3 Análisis del código

En este apartado se va a describir la función de cada uno de los archivos de código de este paquete.

- **canusb.py:** Este fichero contiene las librerías que permiten desencapsular las tramas CAN recibidas por USB y publicarlas en el *topic canrx*. Además se suscribe al *topic cantx* y encapsula la información a enviar al bajo nivel mediante USB.

- **lectura.py**: Este fichero se suscribe al *topic canrx* y demultiplexa la información en diferentes *topics* en función del nodo de procedencia. La información de cada uno de los *topic* se ha formateado con el mensaje más adecuado en función del tipo de dato.

- **test.py**: Este fichero lanza una interface gráfica, que permite ver de forma sencilla la información de los sensores de ultra sonido, así como enviar comandos de movimiento a las ruedas, ver imagen 3.6. Para ello se suscribe al *topic canrx* para obtener la información de los sensores y publica los comandos de velocidad en el *topic cantx*.

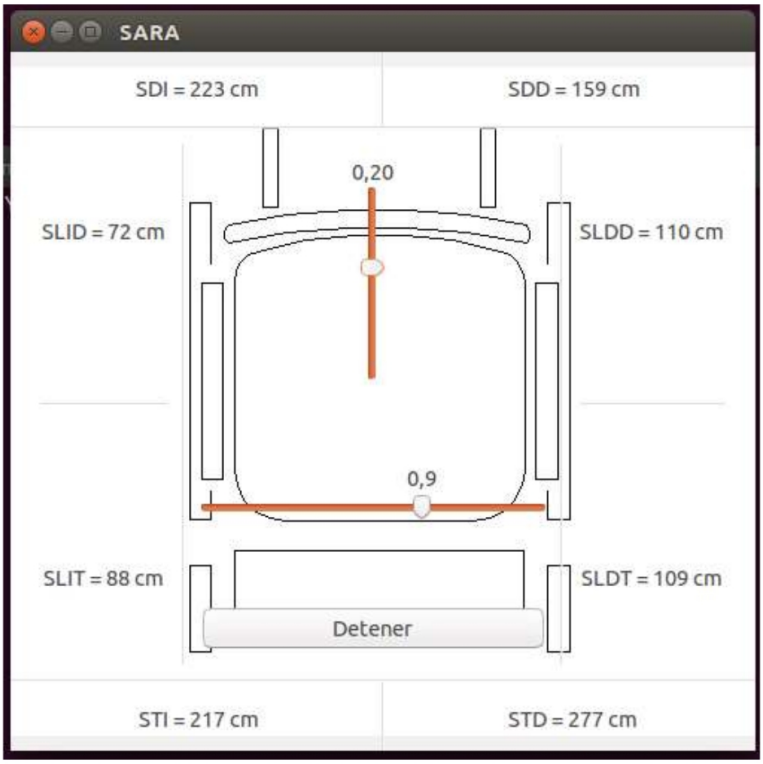


Figura 3.6: Interface gráfica.

3.3.4 Modificaciones implementadas

El TFG de partida [1], implementó el paquete *canusb* con los tres ficheros descritos en el apartado anterior. En este trabajo se ha modificado el fichero **lectura.py**, para sustituir los mensajes genéricos tipo *Int16* por los mensajes estándar de ROS más adecuados según el tipo de dato.

- Para enviar la información de los sensores se ha usado el mensaje *Range.msg* Este mensaje contiene la información de la medición de distancia en metros,la distancia mínima y máxima medible por el sensor y además una marca temporal.

```
# Single range reading from an active ranger that emits energy and reports
# one range reading that is valid along an arc at the distance measured.
# This message is  not appropriate for laser scanners. See the LaserScan
# message if you are working with a laser scanner.

# This message also can represent a fixed-distance (binary) ranger.  This
# sensor will have min_range===max_range===distance of detection.
# These sensors follow REP 117 and will output -Inf if the object is detected
```

```
# and +Inf if the object is outside of the detection range.

Header header           # timestamp in the header is the time the ranger
                        # returned the distance reading

# Radiation type enums
# If you want a value added to this list, send an email to the ros-users list
uint8 ULTRASOUND=0
uint8 INFRARED=1

uint8 radiation_type    # the type of radiation used by the sensor
                        # (sound, IR, etc) [enum]

float32 field_of_view   # the size of the arc that the distance reading is
                        # valid for [rad]
                        # the object causing the range reading may have
                        # been anywhere within -field_of_view/2 and
                        # field_of_view/2 at the measured range.
                        # 0 angle corresponds to the x-axis of the sensor.

float32 min_range       # minimum range value [m]
float32 max_range       # maximum range value [m]
                        # Fixed distance rangers require min_range==max_range

float32 range           # range data [m]
                        # (Note: values < range_min or > range_max
                        # should be discarded)
                        # Fixed distance rangers only output -Inf or +Inf.
                        # -Inf represents a detection within fixed distance.
                        # (Detection too close to the sensor to quantify)
                        # +Inf represents no detection within the fixed distance.
                        # (Object out of range)
```

- Para transmitir la información de los encoders se ha creado un mensaje nuevo, ya que no existe ningún mensaje estándar para este cometido en ROS. Este mensaje contiene la información de las cuentas del encoder, la marca temporal en el momento que se hace la lectura del bajo nivel, un ID para saber con qué encoder de cada rueda se corresponde la información y una marca temporal cuando se recibe el mensaje en ROS.

```
time timestamp
uint16 stdId
int32 extId
uint8[] data
```

- Para enviar la información del sensor inercial se ha usado el mensaje Imu.msg, estandar para este tipo de dato.

```
# This is a message to hold data from an IMU (Inertial Measurement Unit)
#
# Accelerations should be in m/s^2 (not in g's), and rotational velocity should be in rad
    /sec
#
# If the covariance of the measurement is known, it should be filled in (if all you know
    is the
# variance of each measurement, e.g. from the datasheet, just put those along the
    diagonal)
```

```
# A covariance matrix of all zeros will be interpreted as "covariance unknown", and to
    use the
# data a covariance will have to be assumed or gotten from some other source
#
# If you have no estimate for one of the data elements (e.g. your IMU doesn't produce an
    orientation
# estimate), please set element 0 of the associated covariance matrix to -1
# If you are interpreting this message, please check for a value of -1 in the first
    element of each
# covariance matrix, and disregard the associated estimate.

Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y z
```

Además se ha creado el fichero *canusb.launch*, que permite lanzar los nodos de forma automática con los parámetros necesarios para su funcionamiento. Además este formato de fichero, permite anidar el lanzamiento con otros ficheros *.launch*, que se usarán posteriormente para lanzar el resto de paquetes del sistema de navegación.

```
<launch>

  <master auto="start"/>

  <node name="canusb" pkg="canusb" type="canusb.py" output="screen">
    <param name="port" value="/dev/ttyUSB0" />
    <param name="baud" value="1m" />  <!-- baud = rospy.get_param('~baud') -->
  </node>

  <node name="lectura" pkg="canusb" type="lectura.py" output="screen"/>


</launch>
```

3.4 Modelo de la silla (URDF)

3.4.1 Introducción

La silla de ruedas dispone un modelo 3D creado por David Pinedo en un TFG anterior [11], que permite ser usado por ROS en múltiples utilidades, como se mostrará más adelante.

En la imagen 3.7 se muestra el modelo descrito.

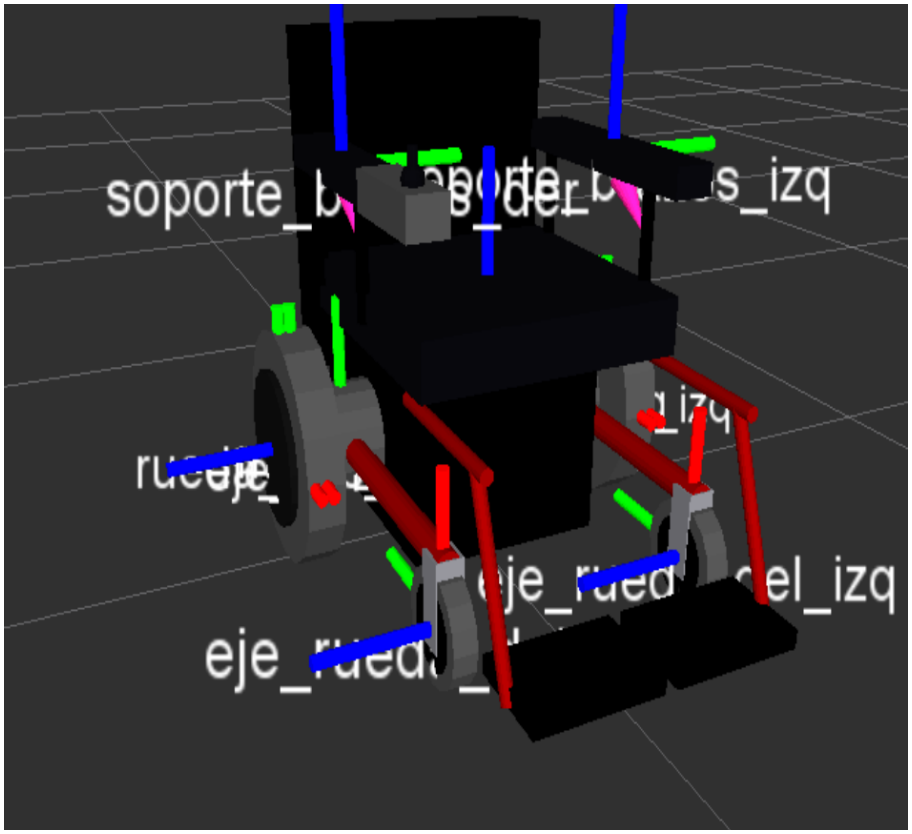


Figura 3.7: Modelo de la silla de ruedas creado por David Pinedo.

3.4.2 Características

El modelo se ha creado en formato URDF (Unified Robot Description Format) [12], que es la especificación XML para describir un el modelo cinemático 3D de un robot que se usa en ROS.

La descripción de un robot en formato URDF consiste en un conjunto de *links*, que definen un cuerpo rígido con inercia y ciertas características visuales, y un conjunto de *joints*, que conectan los *links* entre sí y definen las características cinemáticas y dinámicas de la articulación, además de limitar los posibles giros y movimientos.

La descripción URDF típica de un robot tiene el siguiente formato:

```
< robot name=?ejemplo?>
<link1> ... </link1>
<link2> ... </link2>
<joint> ... </joint>
</robot>
```

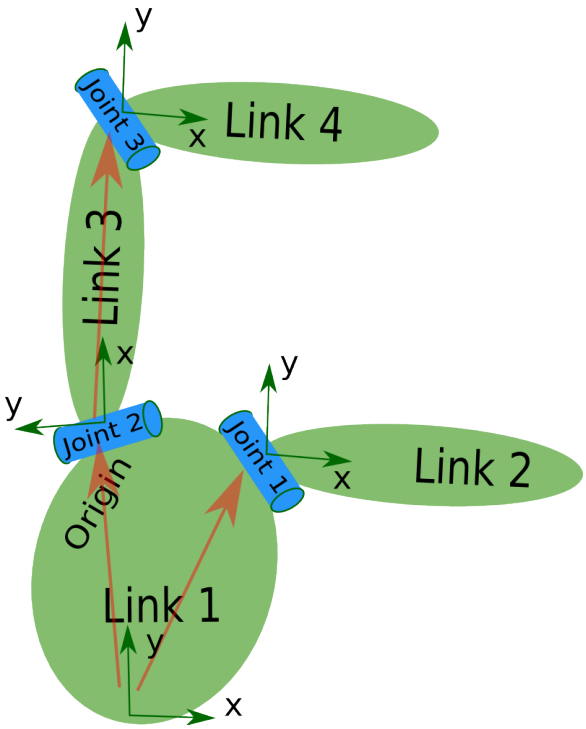


Figura 3.8: Ejemplo de estructura URDF.

En la Figura 3.8 se muestra un ejemplo de una posible estructura de árbol, formada por cuatro *links* con sus tres *joints*.

A la hora de crear un modelo URDF complejo, con un gran número de *links* y de *joints*, el código puede crecer demasiado y volverse complicado de entender. Para evitar ese problema se usa el lenguaje *XACRO* [13], que permite crear archivos XML más cortos y más legibles mediante el uso de macros.

3.4.3 Aplicaciones

En este trabajo, el modelo URDF será usado junto con la interface gráfica *RVIZ* (descrita en el apartado 4.4), para mostrar el modelo de la silla en la representación 3D de la navegación. También es posible usar este modelo en el simulador Gazebo [14].

Capítulo 4

Desarrollo del sistema de control y navegación

4.1 Control (*sara_control*)

4.1.1 Introducción

En este apartado se va a describir el sistema de control diseñado y porqué se ha optado por implementarlo con el paquete *ros control*, que permite hacer un control de alto nivel de la velocidad lineal (V) y angular (Ω) de la silla.

El *hardware* de la silla dispone de un control PID que permite hacer un control de bajo nivel de la velocidad angular (w) de cada una de las ruedas.

En una primera aproximación se utilizó únicamente este control PID, pero los resultados no fueron satisfactorios ya que el módulo de navegación requiere que el sistema tenga unas especificaciones dinámicas concretas, tales como una velocidad lineal (V) y angular (Ω) máxima, lo cual el control PID de la velocidad angular de las ruedas (w) no proporciona.

Además es necesario un control de la cinemática no holonómica de la silla de ruedas, que permita controlar la relación de la velocidad angular de las dos ruedas motrices al introducir consignas de velocidad lineal (V) y angular (Ω).

Para cubrir las necesidades mencionadas, se ha usado el paquete de ROS *ros control* que complementa al paquete de navegación y además permite definir ciertas características dinámicas.

En el listado 4.1 se muestra un extracto de los parámetros que se pueden aplicar al controlador de alto nivel.

Listado 4.1: Extracto de parámetros de ros control.

```
linear:
  x:
    has_velocity_limits    : true
    max_velocity           : 0.6  # m/s
    has_acceleration_limits: true
    max_acceleration       : 0.4  # m/s^2
    min_acceleration       : -0.4 # m/s^2
    has_jerk_limits        : true
```

```
        max_jerk                : 1.0  # m/s^3
```

En la imagen 4.1 se muestra cómo se debe integrar el sistema de navegación de ROS, intercalando *ros control* entre el paquete de navegación y el *hardware*. Concretamente, se muestra la implementación de una base móvil con el controlador *base_controller* y un brazo robótico, con el controlador *arm_controller*.

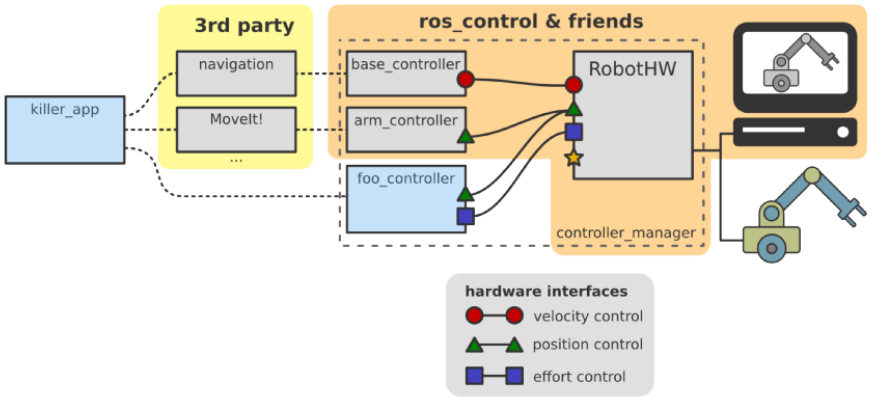


Figura 4.1: *Ros control* junto con *ros navigation* [5].

4.1.2 Controlador de bajo nivel: PID

En este apartado se va a describir el funcionamiento del control PID de velocidad angular w de las ruedas de la silla. Este control permite corregir las perturbaciones debidas al par de carga y debidas a la variación de tensión de la batería

EL controlador está integrado en la tarjeta de control de motores AX3500 7.1, en la cual se han configurado los siguientes parámetros: (Más información en[6]).

- Kp, Ganancia Proporcional = 2,0.
- Ki, Ganancia Integral = 1,5.
- Kd, Ganancia Diferencial = 0,0.

Puesto que se trata de un control PI sobre una planta de tipo 0 el sistema controlador resultante es de tipo 1 por lo que el error de seguimiento es nulo ante entrada escalón. En el apartado 5.3 se analiza el comportamiento del controlador PID.

El sistema implementado en la tarjeta de control en un trabajo anterior [6], sigue el diagrama de la figura 4.2. Por desgracia, las constantes definidas para el controlador PID tienen un valor que ha resultado incompleto en las pruebas realizadas en este trabajo, pues no se definía la relación entre las velocidades angulares en rad/s y los datos que se envían y reciben del bus CAN. Por lo que ha sido necesario calcularlas experimentalmente.

El resultado de la reidentificación de estas constantes ha sido:

$$k_{dato} = 7{,}8125 \frac{dato}{rad}$$

$$k_{encoder} = 64000 \frac{rad}{cuenta}$$

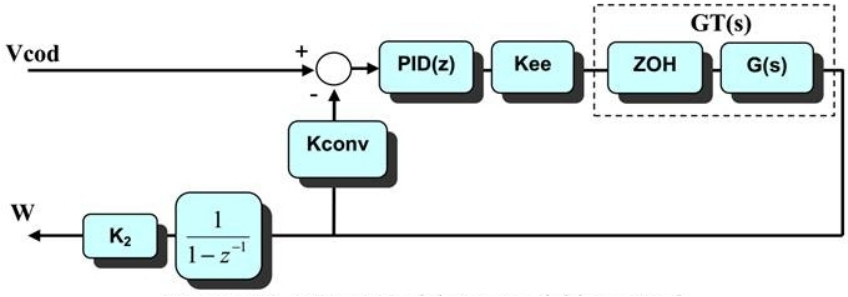


Figura 4.2: Diagrama de bloques del control PID [6].

Estas constantes aparecen en el nuevo diagrama de bloques simplificado del controlador PID de la imagen 4.3.

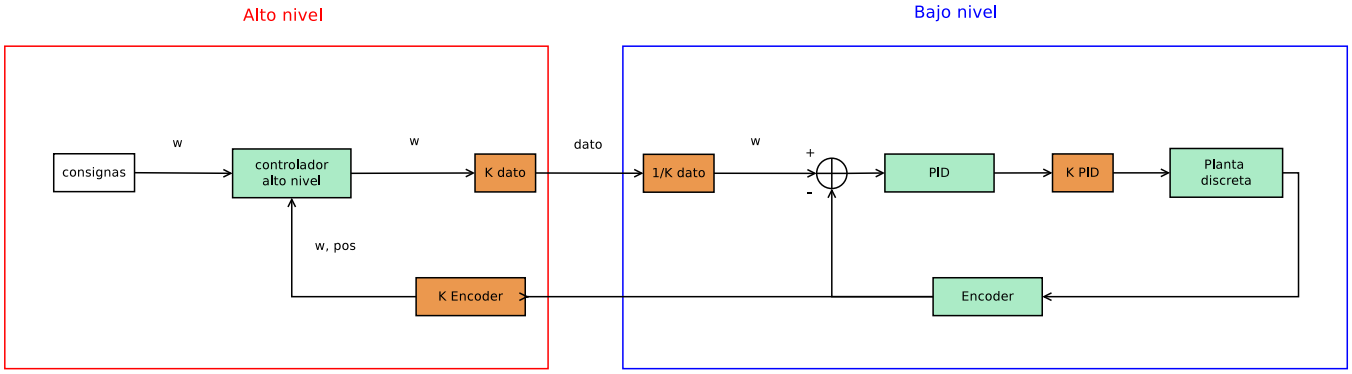


Figura 4.3: Diagrama PID.

Para activar el controlador es necesario poner el carácter *C* en mensaje de velocidad en el apartado *LAZO*, ver 3.1.

4.1.3 Controlador de alto nivel (*ros_control*)

4.1.3.1 Introducción

Ros_control, es un conjunto de paquetes que permiten implementar un sistema de control de forma sencilla, sin necesidad de modelar la planta ni conocer en profundidad la teoría de control. Esto permite ahorrar una gran cantidad de tiempo y obtener una gran fiabilidad en las estrategias de control al usar los controladores desarrollados y probados por la comunidad de desarrolladores de ROS.

En este proyecto se ha usado el controlador *diff_drive_controller*, destinado a robots de tipo *differential drive* como es el caso de la silla de ruedas de este trabajo. Este controlador permite fijar el comportamiento dinámico del robot, además genera la cinemática inversa y el *dead reckonig* descritos en el apartado 2.3.

El paquete ros_control está compuesto de varios bloques como se aprecia en la imagen 4.4:

- **Controller Manager** (En azul en la imagen 4.4): Este paquete permite gestionar los controladores que se ejecutan según la aplicación. En este proyecto se usa el controlador *diff_drive_controller*.
- **Hardware Resource Interface Layer** (En gris en la imagen 4.4): Se trata de la capa que hace de *interface* entre el controlador y *hardware_interface*. Esta capa se declara en el fichero *sara_hw_interface.h* que se describe más adelante en el aparado 4.1.3.3.

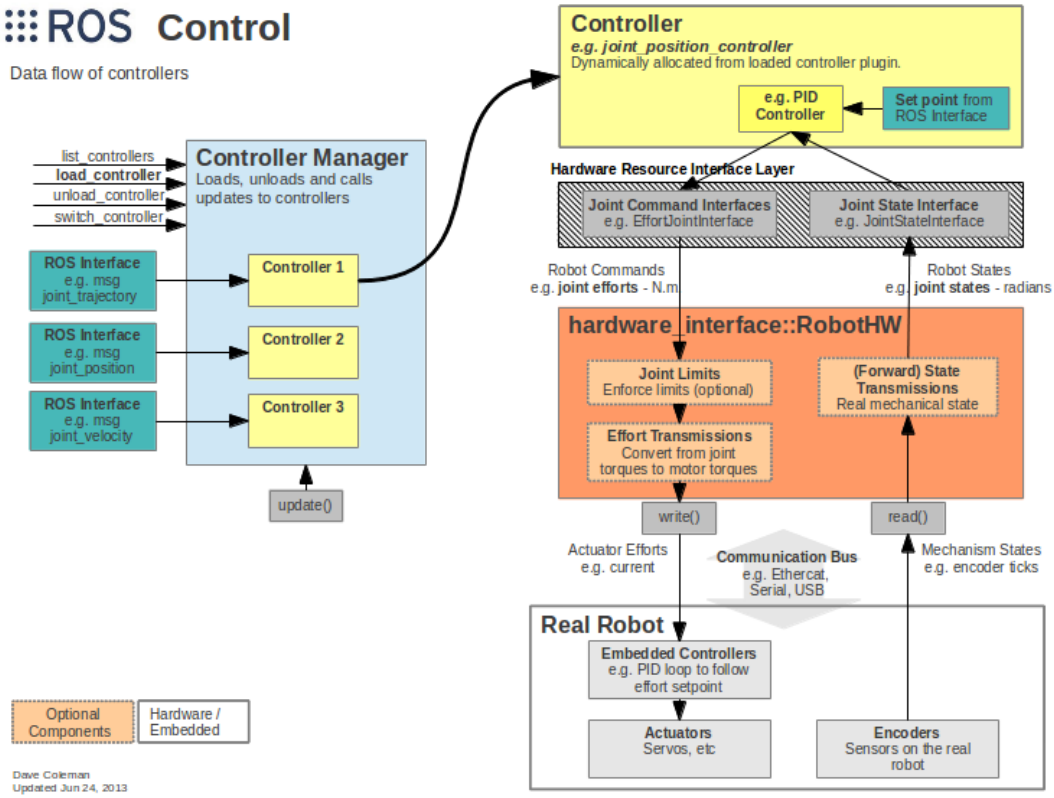


Figura 4.4: Esquema funcional de ROS control [5].

- **Hardware interface** (En naranja en la imagen 4.4): Se trata del bloque que intercambia información el robot y *Hardware Resource Interface Layer*. Se describe en el apartado 4.1.3.3.
- **Real Robot**: Bloque que recibe las consignas y devuelve el estado. Representa la silla de ruedas en nuestro caso.

El esquema general del sistema de control de alto nivel aplicado a la silla de ruedas se muestra en la imagen 4.5. En los siguientes apartados se describirán los bloques nombrados en el párrafo anterior.

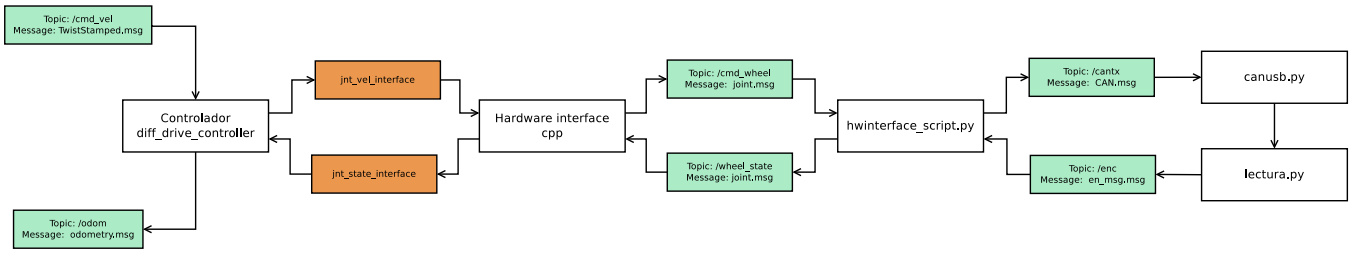


Figura 4.5: Esquema general del control de alto nivel.

4.1.3.2 Controlador (*diff_drive_controller*)

Como se ha comentado en el apartado anterior, para este trabajo se ha usado el el controlador *diff_drive_controller* que permite fijar una serie de parámetros para ajustar el comportamiento diná-

mico de un robot diferencial. Además genera la cinemática inversa y el *dead reckonig* por lo que necesita los datos del radio de la rueda y la separación entre las dos ruedas.

Todos estos parámetros se introducen en el fichero de configuración *diff_drive_controller_paramsyaml* dentro del paquete *sara_control*. A continuación se muestran varios segmentos de este fichero.

- Diámetro y separación de las ruedas, listado 4.2.

Listado 4.2: Configuración *diff_drive_controller*

```
wheel_separation : 0.525 #m
wheel_radius : 0.161     #m
```

- Parámetros dinámicos, listado 4.3.

Listado 4.3: Configuración *diff_drive_controller*

```
linear:
    x:
      has_velocity_limits    : true
      max_velocity           : 0.6  # m/s
      min_velocity           : -0.1 # m/s
      has_acceleration_limits: true
      max_acceleration       : 0.4  # m/s^2
      min_acceleration       : -0.4 # m/s^2
      has_jerk_limits        : true
      max_jerk               : 1.0  # m/s^3
  angular:
    z:
      has_velocity_limits    : true
      max_velocity           : 3.0  # rad/s
      has_acceleration_limits: true
      max_acceleration       : 1.0  # rad/s^2
      has_jerk_limits        : true
      max_jerk               : 2.5  # rad/s^3
```

- Llamada al controlador (*type*), definición del nombre de las uniones de las ruedas (ver 4.1.3.3) y velocidad de publicación de la odometría, listado 4.4.

Listado 4.4: Configuración *diff_drive_controller*

```
  type        : "diff_drive_controller/DiffDriveController"
  left_wheel  : 'wheel_left_joint'
  right_wheel : 'wheel_right_joint'
  publish_rate: 50.0     #odometry
```

Este controlador dispone de las siguientes entradas y salidas. (Ver imagen 4.5)

- **Entradas**
  - *cmd_vel*: Se trata de un *topic* con un mensaje tipo *Twist*. El controlador se suscribe a este *topic* para leer las consignas de velocidad lineal (v) y velocidad angular (Ω), creadas por el sistema de navegación.
  - *joint_states*: Estado de las ruedas del robot (w), se proporciona a través de *hardware_interface* con un tipo de datos especial llamado *joint*. (Para más información ver el apartado 4.1.3.3).

- **Salidas**
  - *odom*: Se trata de un *topic* con un mensaje tipo *Odometry* donde publica la velocidad y la posición de la silla, aplicando la cinemática inversa y el *dead reckonig*. Este *topic* será necesario enviarlo al paquete de navegación.
  - *tf*: Este controlador realiza la transformación entre el frame de la silla (base_link) y el frame *odom*. (Para más información ver 4.3)
  - joint_commands: Los comandos de velocidad obtenidos después de procesar el lazo de control se leen en la Hardware_interface. Estos son publicados con un tipo de datos especial llamado joint y posteriormente enviados a la silla de ruedas. Para más información ver el apartado 4.1.3.3.

4.1.3.3 *hardware_interface*

El paquete *ros_control* está diseñado de forma que pueda ser usado para cualquier tipo de robot, por lo que es completamente genérico.

Para poder usar el controlador en la silla de ruedas autónoma es necesario crear una *interface*, con una estructura definida. Además se debe de crear un bucle de control que actualice el lazo de control. Este código debe ser creado en lenguaje c++.

En internet existen algunos ejemplos de cómo se debe de crear la interface pero ninguno está explicado de forma clara. En este TFG se intentará explicar cómo debe de crearse.

La *interface hardware* se compone de tres ficheros:

- **sara_hw_interface.h:** En este fichero de tipo cabecera se define la clase que se encarga de hacer la interface *hardware*.
  - En primer lugar se definen los métodos necesarios para el intercambio de datos entre el *hardware* y el controlador (listado 4.5).
  - Además es necesario declarar dos atributos, que indican al controlador la información de los actuadores, en nuestro caso son los dos motores.

    En primer lugar se define **jnt_state_interface** que proporcionará información de realimentación de los actuadores.

    A continuación se define **jnt_vel_interface** que indica al controlador donde tiene que publicar la salida del lazo de control.

    En nuestro caso la salida es de tipo velocidad, por lo que la variable es definida como VelocityJointInterface. Si el actuador fuera un servomotor, la salida seria de tipo position por lo que la variable sería de tipo PositionJointInterface.

Listado 4.5: Definición de la clase.

```
class MyRobot : public hardware_interface::RobotHW
{
public:
  MyRobot(); //Constructor

  void read(void);   // Read data from hardware here. joint_state
  void write(void);  // Write data to hardware here. joint_command Publication
  int compute_period(void);
```

```
  void vel_Callback(const sensor_msgs::JointState::ConstPtr& msg);
  void setup(MyRobot*);


private:
  hardware_interface::JointStateInterface jnt_state_interface;
  hardware_interface::VelocityJointInterface jnt_vel_interface;

  ros::Publisher cmd_pub;
  ros::NodeHandle n;
};
```

- **sara_hw_interface.cpp:** En este fichero de código fuente se declaran los métodos definidos en el fichero anterior.

  – Método *MyRobot()*: Se trata del constructor de la clase. Se va a usar para configurar los atributos **jnt_state_interface** y **jnt_vel_interface**.

Listado 4.6: Creación de joint_state

```
hardware_interface::JointStateHandle state_handle_a("wheel_left_joint", &control_data
    .pos[LEFT], &control_data.vel[LEFT], &control_data.eff[LEFT]);
jnt_state_interface.registerHandle(state_handle_a);
```

En el listado 4.6 se indica al controlador los parámetros de realimentación para la rueda izquierda. En primer lugar el identificador del actuador *wheel_left_joint*.

A continuación se indica en que variable se encuentra la información de la posición, velocidad y effort que proporciona el actuador. En el caso de este TFG, sólo tenemos información de la posición y la velocidad.

Finalmente se registra en el atributo completo con todas las especificaciones (*registerHandle*).

Listado 4.7: Creación de joint_vel

```
hardware_interface::JointHandle vel_handle_a(jnt_state_interface.getHandle("
    wheel_left_joint"), &control_data.cmd[LEFT]); //Desired command variable
jnt_vel_interface.registerHandle(vel_handle_a);
```

En el listado 4.7 se indica al controlador donde debe publicar la salida de control. En primer lugar se especifica el identificador del actuador, en nuestro caso la rueda de la silla *wheel_left_joint*.

A continuación se indica en que variable se escribe la salida de control *control_data.cmd[LEFT]*.

Finalmente se registra en el atributo completo con todas las especificaciones (*registerHandle*).

– Método *setup*: Se utiliza para formatear las variables.

– Método *read*: Se utiliza para copiar los datos temporales recibidos en el *callback* y copiarlos a la variable que va a leer el controlador. También se utiliza para calcular el retraso entre la llegada de los datos de realimentación y el momento el el que se procesan. Se usará en el método *compute_period*.

– Método *write*: Publica en el *topic cmd_pub* la salida del controlador.

- Método *compute_period*: Este método permite corregir el problema de sincronización que existe entre el bajo nivel y el alto nivel debido a que no existe un reloj común. Este tema se trata con más detalle en el capitulo 4.1.6
- Método *vel_Callback*: En este método se recibe la información de velocidad angular procedente de los encoders mediante el *topic /wheel_state*.

- **sara_control_loop.cpp**: En este fichero se encuentra la función principal donde se ejecuta el bucle de control. A continuación se describen los puntos más importantes del código.

Listado 4.8: Bucle de control.

```
  while(1)
  {

    elapsed_time=ros::Time::now()-last_time;
    last_time=ros::Time::now();

    // Error check cycle time
    const double cycle_time_error = (elapsed_time - desired_update_freq).toSec();
    if (cycle_time_error > cycle_time_error_threshold)
    {
      ROS_WARN_STREAM_NAMED("bucle_control", "Cycle time exceeded error threshold by: "
                                      << cycle_time_error << ", cycle time: " <<
                                          elapsed_time
                                      << ", threshold: " << cycle_time_error_threshold);
    }
    robot->read();
    cm->update(ros::Time::now(), elapsed_time);
    robot->write();
#ifdef ADJUST_RATE
    ros::Rate fixed_rate(robot->compute_period());
    fixed_rate.sleep();
#else
    rate.sleep();
#endif
  }


}
```

En el fragmento de código 4.8 se encuentra el bucle de control.

- En primer lugar se registra periodo entre dos bucles. Usado para comprobar que no se pierde la frecuencia de actualización deseada.(Listado 4.9).

Listado 4.9: Bucle de control.

```
    elapsed_time=ros::Time::now()-last_time;
    last_time=ros::Time::now();
```

- A continuación se hace un test del tiempo entre dos bucles. En caso de ser mayor que el límite se envía un warning. (Listado 4.10).

Listado 4.10: Bucle de control.

```
// Error check cycle time
    const double cycle_time_error = (elapsed_time - desired_update_freq).toSec();
    if (cycle_time_error > cycle_time_error_threshold)
    {
      ROS_WARN_STREAM_NAMED("bucle_control", "Cycle time exceeded error threshold by:
           "
                                      << cycle_time_error << ", cycle time: " <<
                                          elapsed_time
                                      << ", threshold: " <<
                                          cycle_time_error_threshold);
    }
```

- Después se ejecuta el método read(), descrito en el apartado anterior.
- La línea más importante es *cm− > update* donde se ejecuta el algoritmo de control.
- Para terminar, con el método write() se publica en el *topic /cmd_wheel* la salida del controlador.
- El último bloque es la llamada al método compute_period que permite ajustar los problemas de sincronización. Este tema se trata con más detalle en el capítulo 4.1.6

Listado 4.11: Creación del objeto controller manager.

```
controller_manager::ControllerManager cm(&robot);
```

En la línea 4.11 se crea un objeto de tipo ControllerManager y se pasa por referencia el objeto de la clase donde se encuentra la interface *hardware robot*. Esto permite cargar en el controlador deseado, en nuestro caso *diff_drive_controller* las interfaces declaradas en la clase.

Listado 4.12: Creación del hilo de control.

```
boost::thread(control_loop, ros::Rate(LOOP_RATE),&robot,&cm);
```

En la línea 4.12 se inicia el hilo donde se ejecuta el lazo de control. El hilo o thread es una herramienta de c++ que permite que una función se ejecute en un hilo independiente, lo que permite que la función principal continue ejecutándose, a pesar de que el lazo de control cuente con un bucle infinito.

Esto es importante ya que si se bloquea el programa principal nunca llegarán las *callback* procedentes de los *topics* a los que se ha suscrito este nodo.

4.1.4 *Interface ros_control-canusb*

La *Hardware interface* (capítulo 4.1.3.3), se ha diseñado de tal forma que la entrada y la salida sea en velocidad angular (rad/s), de esta manera el bloque es modular y permite la implementación en otros robots.

Los mensajes de entrada y salida del hardware de la silla de ruedas autónoma usan estas unidades, por lo tanto es necesario realizar una conversión entre los mensajes de entrada y salida de la silla y los de la interface *hardware*. Esta conversión se realizado en el *script hwinterface_script.py*.

En esta imagen 4.6 se muestra la estructura completa que se ha utilizado para enviar y recibir mensajes de la silla.

A continuación se explican los apartados más importantes del código:

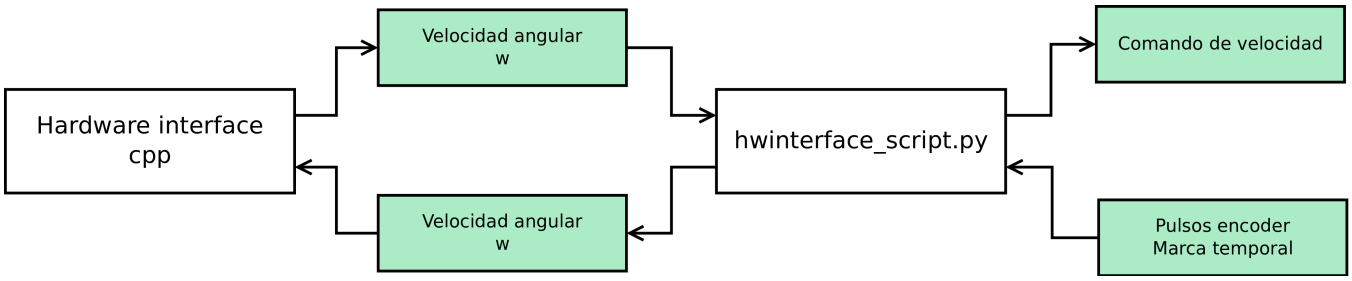


Figura 4.6: Esquema de la interface entre hardware_interface y la silla.

- La función *callback_vel* (listado 4.13), recibe el *topic* con la velocidad angular proveniente de *hardware interface*, para poder convertirlo a los mensajes que acepta la silla. Posteriormente se envían en el *topic cantx* para que el paquete *canusb* lo envíe a la silla.

  El comando de velocidad tiene que estar entre (1-127) para velocidades positivas y entre (255-129) para velocidades negativas.

  Por seguridad solo se envían comandos cuando se ha pulsado el botón 3 (*self.modoPC==3*) en el joystick de la silla de ruedas autónoma (ver 3.2.5 apartado modoPC)

Listado 4.13: Recepción de comandos de velocidad angular.

```
def callback_vel(self,msg): #Recepcion de velocidades

    #Guardar variables
    self.comandD=msg.velocity[self.right] #rad/s
    self.comandI=msg.velocity[self.left] #rad/s

    #Ganancia

    self.datod=int(self.comandD*self.kdato)
    self.datoi=int(self.comandI*self.kdato)

    #Saturacion para evitar errores
    if self.datod>127:
      self.datod=127
    if self.datod<-127:
      self.datod=-127

    if self.datoi>127:
      self.datoi=127
    if self.datoi<-127:
      self.datoi=-127

    #Ajuste de dato negativo
    if  self.datod<0:
      self.datod=256+self.datod #Maxima velocidad 129, 128 parado

    if  self.datoi<0:
      self.datoi=256+self.datoi

    if(self.modoPC==3): #Sincronizado con la silla,permito enviar datos
      dato = CAN()
      dato.stdId = 288
      dato.extId = -1
      dato.data = struct.pack('B', self.datod) + struct.pack('B', self.datoi) + struct.
          pack('B', 0) + struct.pack('B', 0) + 'C' + struct.pack('B', 0) + struct.pack('B
```

```
          ', 0) + struct.pack('B', 0)

self.canpub.publish(dato)
```

- En la función *callback_odom* (listado 4.14), se recibe la información de los encoders (pulsos y marca temporal) que permite calcular la velocidad angular de las ruedas. Para ello es necesario saber cuantos pulsos se reciben en cada vuelta que en este caso es de 64000, para más información consultar el capítulo 3.2.2.

  Finalmente se envía en el *topic /wheel_state* la información de posición y velocidad de la rueda, requerido por el controlador de alto nivel.

Listado 4.14: Recepción de los datos de odometría.

```
def callback_odom(self,msg):
    if(msg.encID==0): #EncoderA (RIGHT)

      self.time_enc_right=msg.time
      self.steps_enc_right=msg.data

      #Calcular incrementos
      dt_right=(self.time_enc_right-self.time_enc_right_last)*(10**-4) #100us
      dsteps_right=self.steps_enc_right-self.steps_enc_right_last

      #Guardiar variables actuales
      self.time_enc_right_last=self.time_enc_right
      self.steps_enc_right_last=self.steps_enc_right

      #Posicion
      self.posD=(self.steps_enc_right/self.pasos_vuelta)*2*pi
      #Velocidad angular
      self.wd=((dsteps_right/self.pasos_vuelta)*2*pi)/dt_right

      #Save data to publish
      data=JointState()
      data.name= ["RIGHT"]
      data.position=[self.posD]
      data.velocity=[self.wd]
      data.effort=[0]
      data.header.stamp = rospy.Time.from_sec(self.time_enc_right_last*(10**-4)) #rospy.
          Time.now()
data.header.frame_id = "base_link"

#Publish topic
self.data_pub.publish(data)
```

- Para terminar, la función *check* (listado 4.15), se ejecuta de forma periódica para comprobar que se han recibido nuevas consignas de velocidad. En caso negativo se para la silla por seguridad.

Listado 4.15: Recepción de los datos de odometría.

```
def check(self,modo):

      if(modo==3): #Sincronizado con la silla

        if (rospy.get_time() - self.lastTwistTime) > self.twistTimeout: #No he recbido
            datos en un tiempo, para la silla
```

```
            dato = CAN()
            dato.stdId = 288
            dato.extId = -1
            dato.data = struct.pack('B', 0) + struct.pack('B', 0) + struct.pack('B', 0) +
                struct.pack('B', 0) + 'A' + struct.pack('B', 0) + struct.pack('B', 0) +
                struct.pack('B', 0)

            self.canpub.publish(dato)

            #rospy.loginfo_throttle(5, "Parada por no recibir datos")
rospy.loginfo_once("Parada por no recibir datos")
```

4.1.5 Lanzamiento

Para poder lanzar los nodos de *ros_control*, la *Interface Hardware* y el *script* de forma ordenada es necesario usar un fichero *.launch*.

A continuación se muestran las partes más importantes del código:

- En listado 4.16 se lanzan los nodos del *script* y la *interface hardware*.

Listado 4.16: Lanzamiento del script y la interface hardware

```
 <!-- Launch SARA interface -->
  <node name="SARA_interface" pkg="sara_control" type="hwinterface_script.py" output="
      screen"/>

  <!-- Load hardware interface -->
<node name="SARA_control_interface" pkg="sara_control" type="control_node" output="screen
    "/>
```

- A continuación, en el listado 4.17 se cargan los parámetros del controlador que se encuentran en el fichero *diff_driver_controller_params.yaml*.

Listado 4.17: Carga de los parámetros del controlador

```
<!-- Load controller settings -->
<rosparam file="$(find sara_control)/config/diff_driver_controller_params.yaml" command="
    load"/> output="screen"/>
```

- Para terminar se lanza el controlador, listado 4.18. En el argumento se debe poner el mismo texto que se ha usado para nombrar el controlador en el fichero de ajustes que se detalla en el listado 4.3.

Listado 4.18: Carga del controlador

```
<!-- Load controller manager -->
  <node name="SARA_controller_manager" pkg="controller_manager" type="spawner"  output="
      screen"  args="diff_drive_controller" >
</node>
```

4.1.6 Sincronización

En la elaboración de entre trabajo se ha presentado un problema de sincronización entre el *hardware* y el software de alto nivel.

El *hardware* envía una trama CAN con la información de los encoders cada 100ms y el lazo de control se ejecuta cada 100ms, pero debido a que no existe un reloj común existe una pequeña diferencia de frecuencia entre ambos relojes que provoca que en algunos periodos de muestreo no se tenga información del bajo nivel.

En la imagen 4.7, se muestran los tres casos que se pueden dar.

- En el primer periodo de muestreo la información de ambos encoders (flechas roja y azul) llega antes de que se produzca el computado del control de alto nivel. Por lo tanto se realiza correctamente.
- En el segundo periodo, debido al pequeño desfase entre el reloj de alto nivel y el reloj del *hardware*, la información de los encoders no ha llegado antes de que se compute el control de alto nivel, por lo tanto se hace de forma incorrecta.
- En el tercer periodo se muestra otro de los problemas que afectan a este sistema y es que existe *jitter* en la recepción de los mensajes procedentes del bus CAN. La información de ambos encoders se recibe en mensajes diferentes y no siempre llegan en el mismo momento, debido a que el sistema de alto nivel no es en tiempo real. Esto también puede producir que en algunos periodos de muestreo de alto nivel no se disponga de la información de alguno de los encoders.

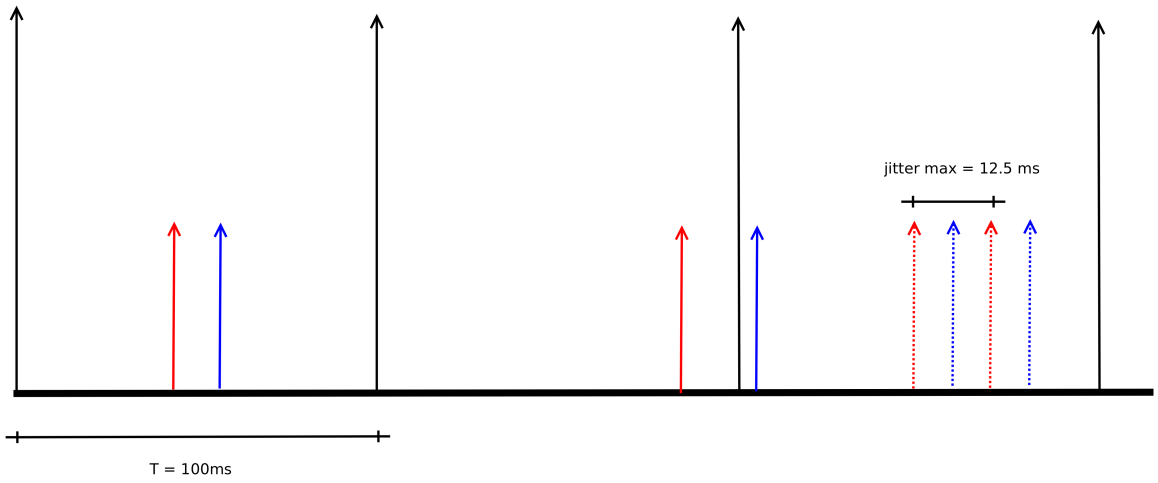


Figura 4.7: Gráfico con los problemas de sincronización entre el alto nivel y el bajo nivel. Las flechas azul y roja representan el momento en el que se recibe la información de los encoders de cada una de las ruedas. Las flechas negras representan el periodo de muestreo del control de alto nivel.

Para mitigar este problema se ha incorporado una función activable en el control de alto nivel, que intenta mantener la diferencia de tiempo mayor entre la llegada de los mensajes de los encoders y el periodo de muestreo del control de alto nivel. Este funcionamiento se muestra en la figura 4.8

- En el primer periodo de muestreo, la diferencia de tiempo entre la llegada de la información de los encoders y el muestreo del control de alto nivel es muy pequeña.
- En el segundo periodo de muestreo se corrige el problema anterior, alargando ligeramente el tiempo de muestreo en el siguiente periodo de forma que la llegada de las muestras coincida con la mitad del periodo, maximizando la diferencia de tiempo y evitando los problemas del apartado anterior.

- En el tercer periodo se muestra que con esta solución, el *jitter* no afecta al funcionamiento del control, ya que este no supera los 12.5ms siendo menor que la diferencia de tiempo con el siguiente periodo de muestreo.

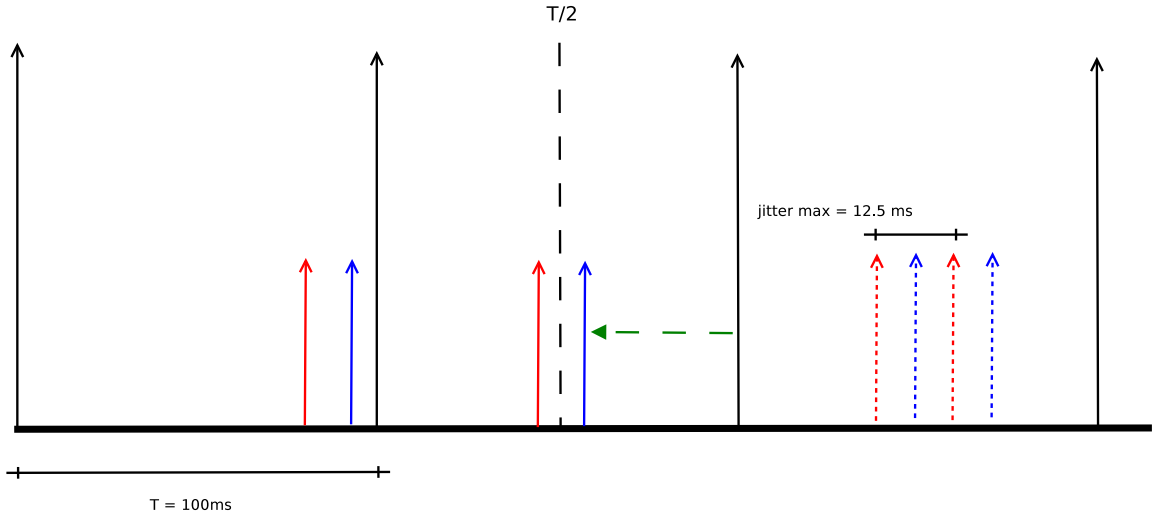
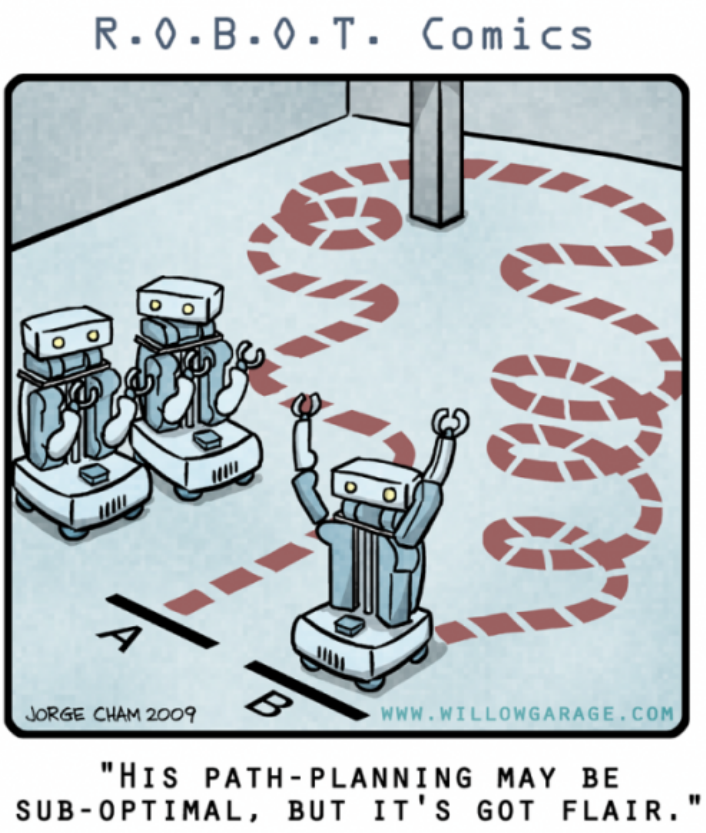


Figura 4.8: Gráfico que muestra el funcionamiento de la función de corrección de la sincronización. Las flechas azul y roja representan el momento en el que se recibe la información de los encoders de cada una de las ruedas. Las flechas negras representan el periodo de muestreo del control de alto nivel.

Esta corrección solo entra en funcionamiento cada varios minutos, por lo que no altera el comportamiento del sistema de control.

4.2 Navegación (*ros_navigation*)



4.2.1 Introducción

El sistema de navegación autónoma de la silla de ruedas, se ha implementado gracias a *ros_navigation* [2], que permite implementar de forma sencilla un sistema de navegación.

En esta capítulo se va analizar el funcionamiento de este sistema y se va a explicar la implementación realizada en la silla de ruedas.

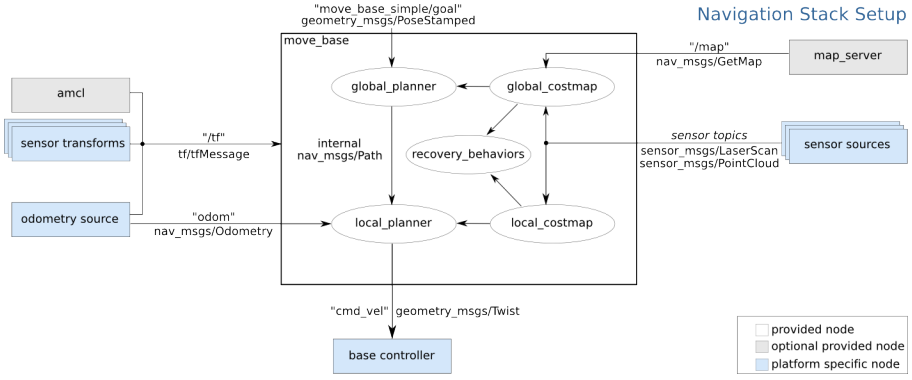


Figura 4.9: *Navigation Stack*

Ros navigation [2] está formado por un gran número de paquetes, pero el núcleo es el *Navigation Stack* (imagen 4.9) que cuenta con todo lo necesario para hacer posible la navegación.

Navigation Stack cuenta con los siguientes bloques:

- *move_base* (recuadrado en la figura 5): Realiza el trabajo principal de navegación.
- *map_server*: Proporciona el mapa al stack.
- *odometry source*: Recibe la posición y velocidad del robot. En este trabajo es generado por el controlador de alto nivel a partir de los encoders de las ruedas.
- sensor *sources*: Es posible usar numerosos tipos de sensores, pero ROS *Navigation* está centrado en el uso de sensores láser.
- *amcl* (*adaptative Mote Carlo localization*): Sistema de ubicación probabilístico, hace uso de los sensores láser. En este trabajo no se ha usado.
- *sensor transform*: Permite saber la relación de posición entre el robot y el mapa. Y entre el robot y los sensores. Ver sección 4.3 para conocer más a fondo el funcionamiento.
- *base controller*: Representa el robot, al que se envían los comandos de velocidad.

4.2.2 *move_base*

4.2.2.1 Introducción

Move_base [15] es un conjunto de paquetes que engloba el funcionamiento principal de ROS *Navigation*. Proporcionando un objetivo (*goal*), intentará alcanzarlo enviando las consignas de velocidad al robot (*base controller*).

Se encarga de enlazar el planificador global y el planificador local para trazar la ruta adecuada (menos coste) usando la información del espacio proporcionada por los *costmap*, global y local para trazar la mejor ruta.

Está formado por los siguientes paquetes:

4.2.2.2 *Costmap*

El paquete *costmap_2d* [16] crea un mapa de ocupación *ocucupancy grid* en 2d a partir de la información recibida por map_server 4.2.3 y los sensores que se hayan instalado, ya sean láser o ultrasonidos.

En la imagen 4.10 se puede ver un ejemplo del mapa de ocupación creado por el paquete *costmap*. Las marcas rojas representan los obstáculos detectados por los sensores. Las marcas azules representan los obstáculos inflados, es decir los lugares que el centro del robot no debe cruzar para evitar una colisión. El color gris se corresponde con la información del mapa. El polígono rojo representa el *footprint* o silueta del robot.

Para evitar una colisión el *footprint* nunca debe cruzar las marcas rojas y el centro del robot nunca debe cruzar las marcas azules.

En *Navigation Stack* existen 2 *Costmap*:

- global_costamp: Está dedicado a realizar el mapa de ocupación de los obstáculos de largo rango. En el caso de este trabajo realiza el mapa de ocupación de los obstáculos proporcionados por el mapa obstáculos del mapa.

  Este mapa tiene un menor periodo de actualización ya que no es crítico a la hora de evadir obstáculos, porque se usa para realizar la ruta global. En el caso de este trabajo el mapa es estático.

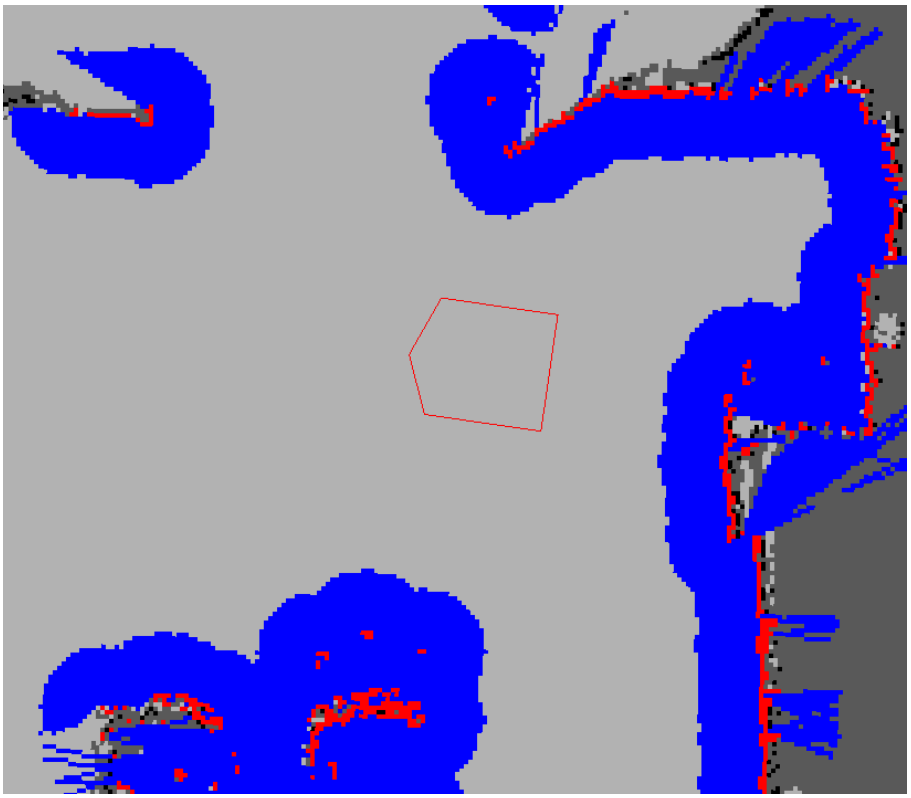


Figura 4.10: Mapa de ocupación creado por *costmap*.

- local_costamp: Está dedicado a realizar el mapa de ocupación de los obstáculos más cercanos. En este trabajo procesa los obstáculos que generan los ultrasonidos.

  Tiene un periodo de actualización rápido para poder evitar los obstáculos detectados.

En la imagen 4.21 se puede apreciar gráficamente los dos *costmap* comentados. La ventana de color blanco representa el *local_costamp*, que rodea el robot y tiene la información del obstáculo mientras que todo el mapa restante se corresponde con el *global_costamp*.

4.2.2.3 *Planner*

En *Navigation Stack* existen dos planificadores de ruta con funcionamientos distintos.

- *Global planner* [17]: El planificador global crea la ruta a largo plazo, sabiendo la posición actual del robot y la posición final.

  Para trazar la ruta usa el mapa de ocupación creado por *global_costmap* que se ha descrito en el apartado anterior. En la imagen 4.11 se muestra un ejemplo de una ruta generada sobre un mapa de ocupación.

  Esta ruta tiene un coste computacional muy bajo ya que no debe actualizarse en tiempo real.

  Existen varios algoritmos para realizar esta ruta, pero se ha usado el algoritmo por defecto ya que los resultados son buenos. Para ver el listado completo de algoritmos, visitar la página del paquete [17].

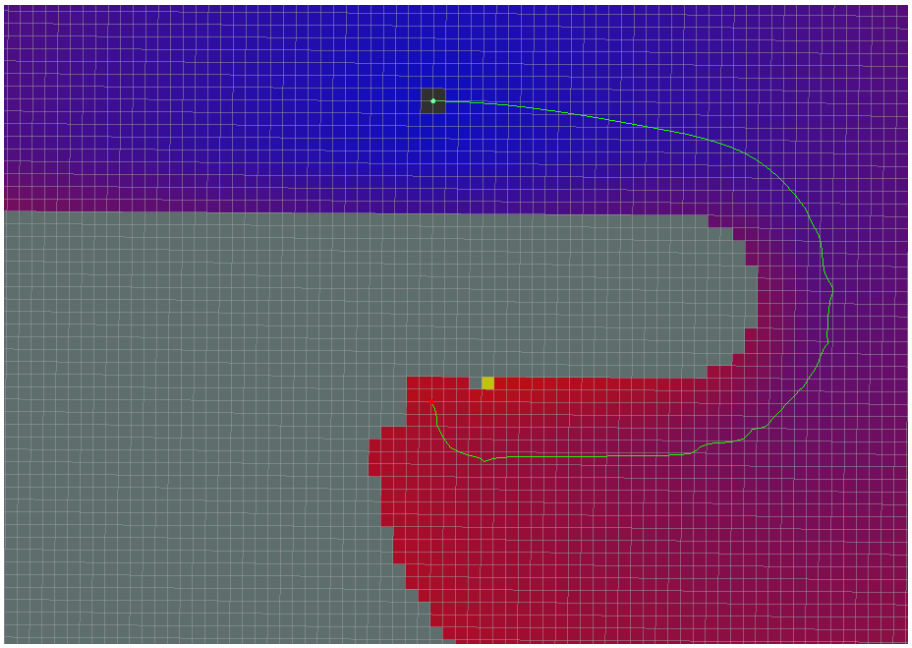


Figura 4.11: Planificador Global.

- *Local planner*: El planificador local crea la ruta que debe seguir el robot a corto plazo.

  En base al estado actual del robot, realiza una simulación que predice el estado futuro del robot en función de las consignas de velocidad y elige la que menor coste tenga, ver imagen 4.12. El algoritmo usado es DWA (*Dynamic Window Approach*), este algoritmo tiene un coste computacional bajo, aunque no funciona bien para aceleraciones muy reducidas. En el caso de este trabajo las aceleraciones son suficientes para usar este algoritmo.

  Para poder generar la ruta, necesita el mapa de ocupación local (*local_costmap*) y haber generado la ruta global.

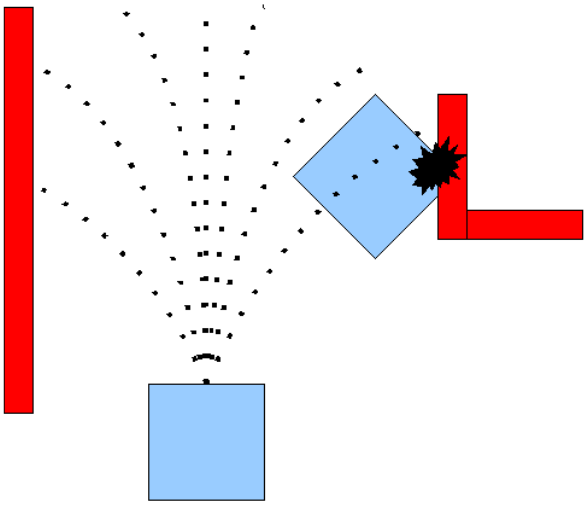


Figura 4.12: Planificador Local.

  Debido a que el planificador local necesita conocer las características dinámicas del robot para realizar la simulación, es necesario implementar un controlador de alto nivel que permita fijar el comportamiento del robot, principalmente fijar la aceleración máxima y mínima. Para ver más información acerca del controlador de alto nivel ver la sección (ros_control 4.1).

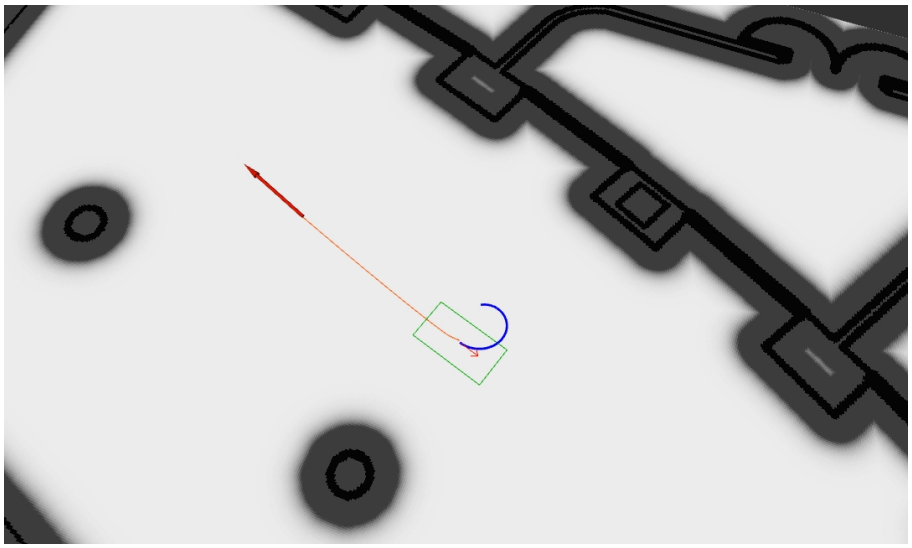


Figura 4.13: Planificador Global y Local.

En la imagen 4.13 se muestra el funcionamiento del Global Plan (rojo), que traza la ruta desde el origen a la meta, y el Local Plan (azul) que envía los comandos de velocidad y traza la ruta que debe seguir el robot.

Se puede ver un ejemplo de funcionamiento aquí https://www.youtube.com/watch?v=Ll5xFwT535o

4.2.2.4 Ficheros de configuración de *move_base*

Cada uno de los paquetes explicados en los apartados anteriores tiene asociado un fichero de configuración. El funcionamiento del robot viene determinado por los ajustes que se van a mostrar a continuación y por eso es necesario realizar un buen ajuste de la gran cantidad de parámetros que tiene.

- Configuración de *local_costmap* y *global_costmap*. La configuración de estos paquetes va a determinar qué información se va a mostrar en cada uno de los mapas de ocupación.

  A partir de la versión Hydro de ros, *costmap_2d* usa *plugins* para configurar las capas del mapa de ocupación. Esta es la configuración del *global_costmap*:

Listado 4.1: Fichero de configuración de *global_costmap*.

```
global_costmap:

  plugins:
    - {name: static_map,       type: "costmap_2d::StaticLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

  global_frame: "map"
  robot_base_frame: "base_link"

  publish_frequency: 50.0
  update_frequency: 3.0

  resolution: 0.5 #0.01 #The resolution of the map in meters/cell.
  transform_tolerance: 0.2 #Specifies the delay in transform (tf) data that is tolerable in
      seconds
  map_type: costmap
```

En la configuración se puede apreciar que se han declarado 2 capas.

- *static_map*: Es la capa que representa la sección del mapa que no cambia. En este trabajo se corresponde con el mapa proporcionado por map_server 4.2.3
- *inflation_layer*: Es la capa que representa los obstáculos del mapa inflados para evitar colisiones.
- Además se han configurado otra serie de parámetros. En *global_costmap* el frame global será el mapa, la frecuencia a la que se publica el mapa es de 50 Hz ya que no supone una carga computacional alta, mientra que la frecuencia de actualización del mapa es baja ya que el mapa es estático y esto supone un impacto significativo en el consumo de recursos. El resto de parámetros se explican en el fichero

El siguiente fragmento de código se corresponde con la configuración del *local_costmap*.

Listado 4.2: Fichero de configuración de *locall_costmap*.

```
local_costmap:
  plugins:
    #- {name: obstacle_layer, type: "costmap_2d::VoxelLayer"} #Laser sensors
    - {name: ultrasonic,   type: "range_sensor_layer::RangeSensorLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

  update_frequency: 5.0
  publish_frequency: 50.0

  global_frame: "odom" #To inflate obstacles
  robot_base_frame: "base_link"

  #static_map: false
  rolling_window: true
  width: 10.0 #6
  height: 10.0 #6
  resolution: 0.05 #0.01
```

En la configuración se puede apreciar que se han declarado 2 capas.

- ultrasonic: Es la capa que permite generar los obstáculos a partir de la información de los sensores. Se explica en el apartado 4.2.5.4.
- inflation_layer: Es la capa que representa los obstáculos del mapa inflados para evitar colisiones. Se usa para inflar los obstáculos generados por los ultrasonidos.
- Además se han configurado otra serie de parámetros. En *local_costmap* el frame global será *odom*, la frecuencia a la que se publica el mapa es de 50 Hz ya que no supone una carga computacional alta, mientra que la frecuencia de actualización del mapa es mayor que en *global_costmap* ya que el mapa se actualiza con la información de los ultrasonidos. El resto de parámetros se explican en el fichero

El siguiente fragmento de código se corresponde con la configuración común de ambos *costmap*.

Listado 4.3: Fichero de configuración común de *costmap*.

```
footprint: [[-0.54 , 0.29], [0.54, 0.29], [0.54, -0.29], [-0.54, -0.29]]

obstacle_layer: #Laser
  obstacle_range: 2.5
  raytrace_range: 3.0
  observation_sources: scan
  scan: {data_type: LaserScan, topic: /scan, marking: true, clearing: true,
      expected_update_rate: 0}


inflation_layer:
  inflation_radius: 1.0

ultrasonic:
  clear_threshold:    0.9
  mark_threshold:     0.95
  no_readings_timeout: 2.0
  clear_on_max_reading: true
  # ns:     /mobile_base/sensors/sonars
  topics: ["/SLIT_range", "/SLDD_range", "/STD_range", "/SDI_range","/SLID_range","/SDD_range"
      ,"/SLDT_range","/STI_range"]
```

En primer lugar se configuran las dimensiones del robot y a continuación se configuran los parámetros de las capas declaradas en los ficheros de configuración de *global_costmap* y *local_costmap*:

- Configuración de *local_planner*

  En este caso no se ha incluido el código de configuración ya que es muy largo, pero todos los parámetros están comentados para facilitar su entendimento. A continuación se describen los puntos más importantes de la configuración.

  - *controller_frequency*: Fija la frecuencia del controlador, que se ha fijado en 10 Hz ya que a una mayor frecuencia el consumo de recursos excede la potencia del ordenador. Además la realimentación de los encoders se realiza a 10 Hz también.
  - Los parámetros dinámicos fijados deben ser iguales a los configurados en el controlador de alto nivel. Se han fijado estos valores ya que resultan confortables para los usuarios de la silla.
  - *holonomic_robot*: Este ajuste se debe de fijar como $false$ ya que la silla de ruedas es un robot no holonómico, que solo avanza en un eje de coordenadas.

– Los parámetros *Forward Simulation Parameters*, determinan cómo se hace la simulación para predecir la futura trayectoria del robot. Estos son los ajustes más críticos y los que más coste computacional conllevan. Los ajustes que se muestran son los que mejor resultado han dado sin superar la capacidad de cómputo del ordenador. Para entender bien estos parámetros se ha usado la siguiente referencia [18].

4.2.3 Mapas (*map_server*)

Existen dos formas de usar los mapas en navegación.

- Usar un mapa predefinido con el paquete *map_server* [19]: El mapa se inserta en una imagen en blanco y negro, que viene acompañada de un fichero *.yaml* que aporta las características (listado 4.19).

Listado 4.19: Fichero de configuración del mapa

```
image: uah_map.png
resolution: 0.02520696
origin: [0, 0, 0]
occupied_thresh: 0.65
free_thresh: 0.196 # Taken from the Willow Garage map in the turtlebot_navigation package
negate: 0
```

  – *image*: Ruta de la imagen
  – *resolution*: Resolución del mapa, metros/píxel.
  – *origin*: Offset de posición del mapa.
  – *occupied_thresh*: Los píxeles con una probabilidad de ocupación mayor que este valor son considerados obstáculos.
  – f*ree_thresh*: Los píxeles con una probabilidad de ocupación menor que este valor son considerados libres de obstáculos.
  – *negate*: Invierte la probabilidad de ocupación

  En este trabajo se ha elegido esta opción y se ha procesado un plano de la UAH para usarlo como mapa y realizar los experimentos prácticos, imagen 4.14.

  La imagen se ha obtenido a partir de los planos en PDF de la escuela Politécnica.

  – En primer lugar el PDF se ha importado a GIMP, con lo se se obtiene una imagen en mapa de bits, con la escala necesaria entre las dimensiones del mapa y los *pixels*. En la imagen 4.15 se muestra la ventana de importación del GIMP, en el apartado *Resolución* aparece la escala de la imagen.
  – A continuación la imagen se ha procesado con MATLAB, aplicando un filtro umbralizador (listado 4.20) que elimina los grises, consiguiendo una imagen binaria (Blanco y Negro), donde el blanco representa espacio libre y el negro representa un obstáculo. Además este procesado reduce en gran medida el tamaño de la imagen y elimina elementos que no son obstáculos.

    El resultado de este filtro aparece en la imagen 4.16. A la izquierda aparece la imagen original y a la derecha aparece la imagen filtrada. Se puede apreciar que gracias al filtrado se han eliminado elementos innecesarios como el tejado de la terraza.

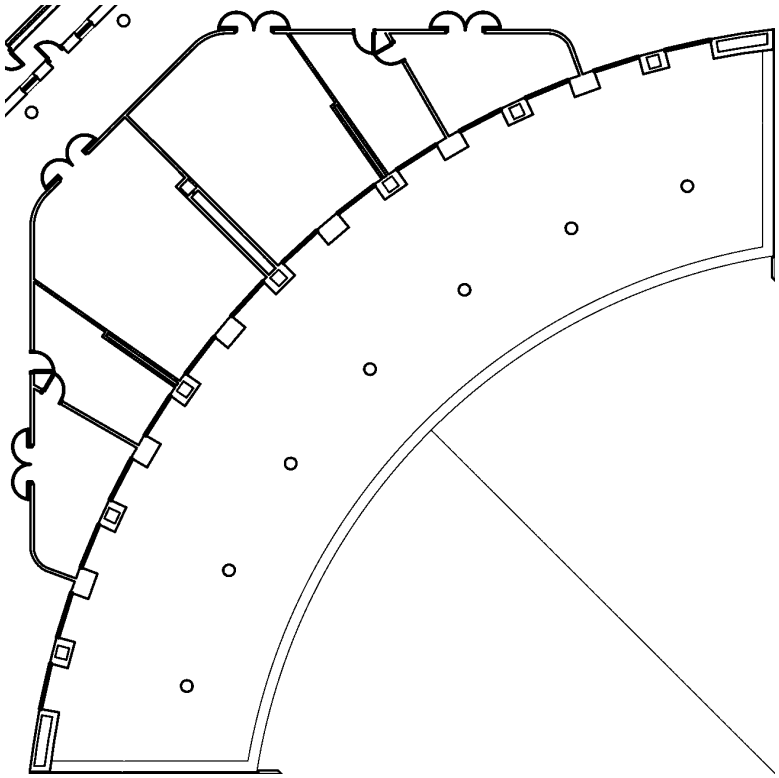


Figura 4.14: Mapa de la UAH procesado.

Listado 4.20: Script de procesado de la imagen en MATLAB

```
A=im2double(rgb2gray(imread('Mapa_10000pxHQ.png')));

umbral = 0.7;
A_umbral=(A>umbral);
imwrite(A_umbral,'filtrada.png')
```

- Crear el mapa a medida que el robot navega *SLAM*, (*simultaneous localization and mapping*): Para realizar este tipo de navegación es necesario contar con un sensor láser que escanee el espacio que le rodea. En este trabajo no se ha realizado este tipo de navegación, pero se puede realizar con el paquete de ROS *gmapping* [20].

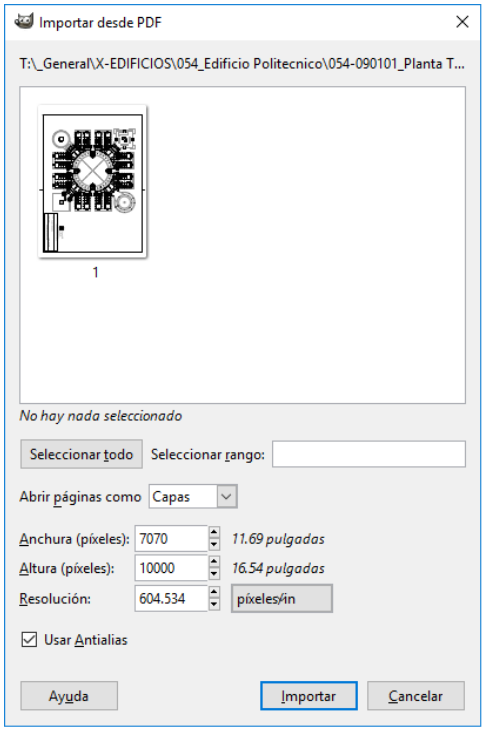


Figura 4.15: Importar el PDF en GIMP.

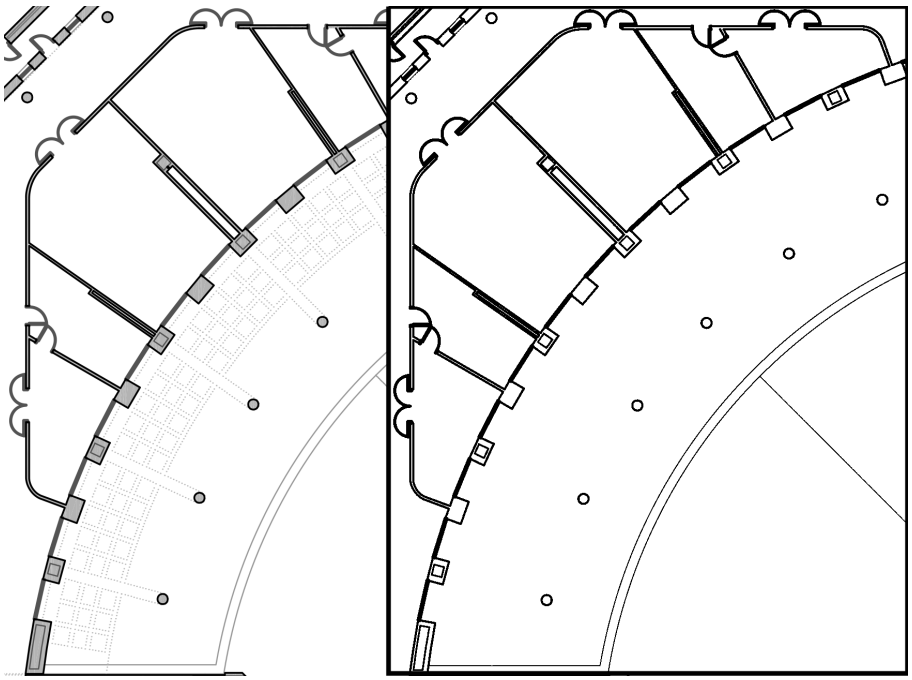


Figura 4.16: Filtrado de la imagen en MATLAB.

4.2.4 *amcl* (*fake_localization*)

amcl es un paquete de ROS que hace uso del algoritmo *Adaptative Mote-Carlo Localization* para realizar una localización probabilística gracias al uso de un sensor láser.

Este paquete crea una nube de probabilidad con las posibles localizaciones del robot como se muestra en la imagen 4.17.

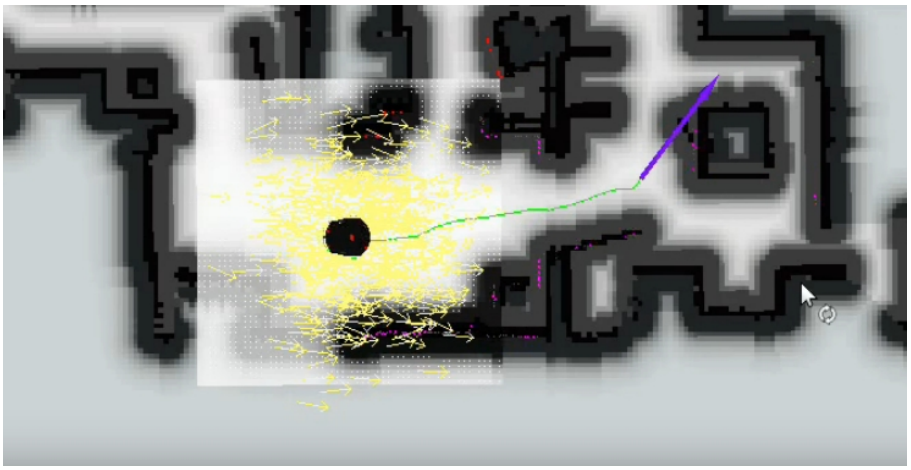


Figura 4.17: Ejemplo de uso de AMCL, momento inicial.

A medida que el robot navega la varianza de la nube es menor como se muestra en la imagen 4.18.

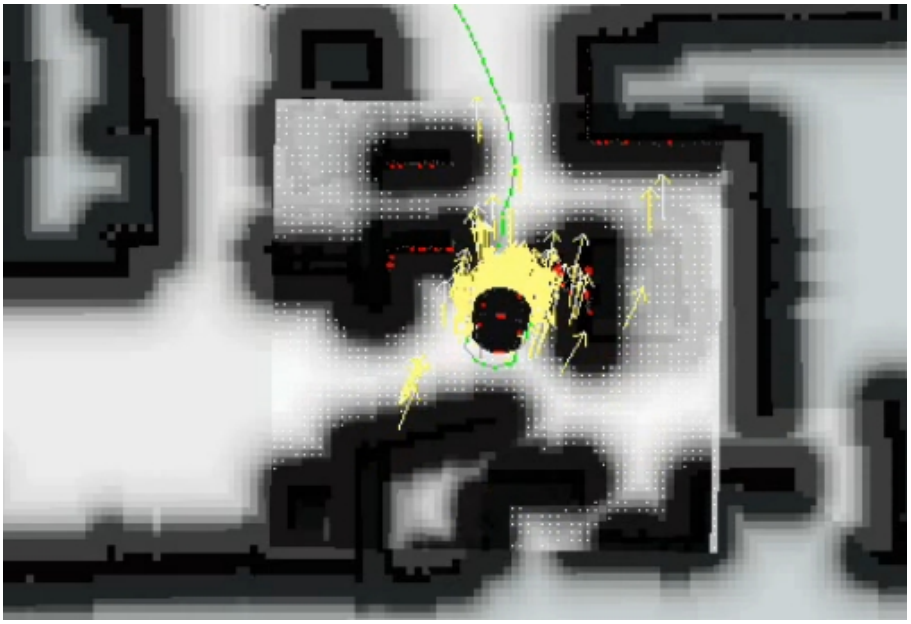


Figura 4.18: Ejemplo de uso de AMCL, despues de haber navegado.

En el caso de este trabajo, no se dispone un sensor láser por lo que no se puede usar este paquete. Por lo tanto es necesario un paquete que use los datos de odometría para realizar la localización del robot.

Afortunadamente existe un paquete que realiza esta función: *fake_localization* [21]. Este paquete se suscribe al *topic* de odometría y publica el *topic amcl_pose* que usa *move_base* para saber la posición del robot.

Al ejecutar este paquete se configuran los siguientes parámetros (listado 4.21):

Listado 4.21: Parámetros de fake_localization

```
<param name="global_frame_id" value="map" />
<param name="odom_frame_id" value="odom" />
<param name="base_frame_id" value="base_link" />
<param name="delta_x" value="-3.4" />
<param name="delta_y" value="-6.5" />
<param name="delta_yaw" value="0.3" />
```

- *global_frame_id*: Frame en el que se publica la transformación global_frame-odom al cambiar la posición inicial.
- *odom_frame_id*: Nombre del frame de la odometría.
- *base_frame_id*: Frame del robot.
- *delta*: Los parámetros delta, permiten fijar la posición inicial del robot antes de la ejecución. Este *offset* se produce entre los frames mapa-odom.

4.2.5 Sesores de ultrasonidos

4.2.5.1 Introducción

La silla cuenta con 9 sensores de ultrasonidos, 8 de ellos están ubicados en los 4 lados de la silla como se muestra en la figura 4.19, mientras que uno se ubica bajo el *joystick*.

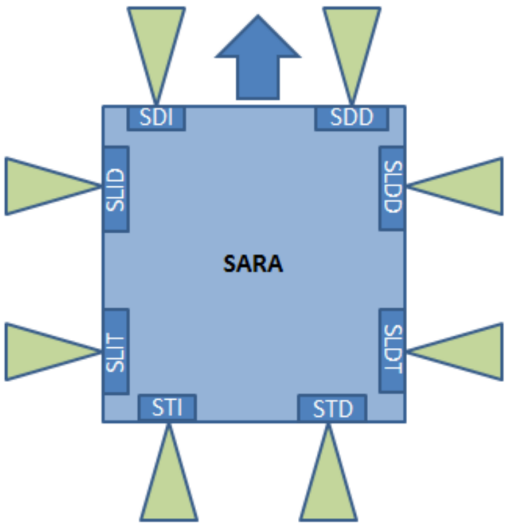
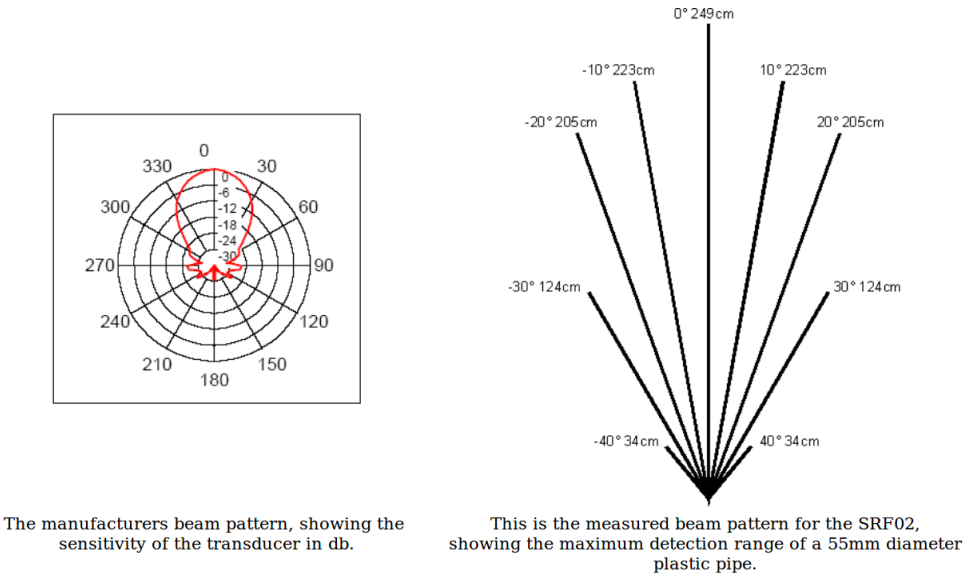


Figura 4.19: Ubicación de los sensores en la silla.

En los siguientes apartados se describe el *hardware* que se ha usado y cómo se ha implementado en el sistema.

4.2.5.2 *Hardware*

Los sensores instalados son el modelo SRF02 que tiene una distancia máxima de detección de 3m y una distancia mínima de 0.01m . Las distancias se han obtenido mediante experimentación ya que el diagrama de detección de la imagen 4.20 no es preciso.



The manufacturers beam pattern, showing the sensitivity of the transducer in db.

This is the measured beam pattern for the SRF02, showing the maximum detection range of a 55mm diameter plastic pipe.

Figura 4.20: Diagrama de radiación de los sensores.

La información de los sensores es enviada a través del bus CAN y se decodifica en el script lectura.py (ver 3.5)

El montaje e integración de los sensores en la silla se describe en el apartado 3.2.3.

4.2.5.3 *Topic*

El *topic* con la información de los sensores es generado en lectura.py y es usado por el paquete de ROS *Navigation* para realizar la evasión de obstáculos. Para ello es necesario usar un mensaje estandart llamado Range.msg (listado 4.22).

Listado 4.22: Mensaje Range

```
std_msgs/Header header
uint8 radiation_type
float32 field_of_view
float32 min_range
float32 max_range
float32 range
```

El *topic* contiene la siguiente información

- *header*: Contiene el *timestamp* y el *frame* asociado al sensor, lo que permite fijar la posición con respecto a la silla.
- *radiation_type*: Define el diagrama de radiación del sensor, puede ser de tipo infrarrojo o ultrasonido.
- *field_of_view*: Ángulo de apertura del diagrama de radiación.
- *min_range y max_range*: Distancia mínima y máxima de funcionamiento del sensor.
- *range*: Distancia en metros del objeto detectado. No puede ser inferior ni superior a los límites establecidos en la línea anterior.

Debido a esta última restricción, es necesario saturar la información de los sensores para que nunca salga del rango especificado, ya que produce el mal funcionamiento de los paquetes que usen estos datos.

Además cuando los sensores detectan un rango infinito, devuelven un valor de 0, que se sustituirá por el objeto $'Inf'$

En el script lectura, donde se generan los *topic* se ha implementado una función que satura la información de los sensores (listado 4.23):

Listado 4.23: Saturación del rango de los sensores

```
def saturate_ultrasonic_data(self,data):

  if data == 0: #Infinite value
    return float('Inf')
  elif data > self.sensor_max_range:
    return self.sensor_max_range
  elif data<self.sensor_min_range:
    return self.sensor_min_range
  else: #In the limits of range
    return data
```

4.2.5.4 Integración con *navigation*

La información de los sensores es usada por el *Local Costmap*, que genera el *ocupancy grid* de los sensores.

Para ello se usa un *plugin* de *costmap* llamado *range_sensor_layer* [22].

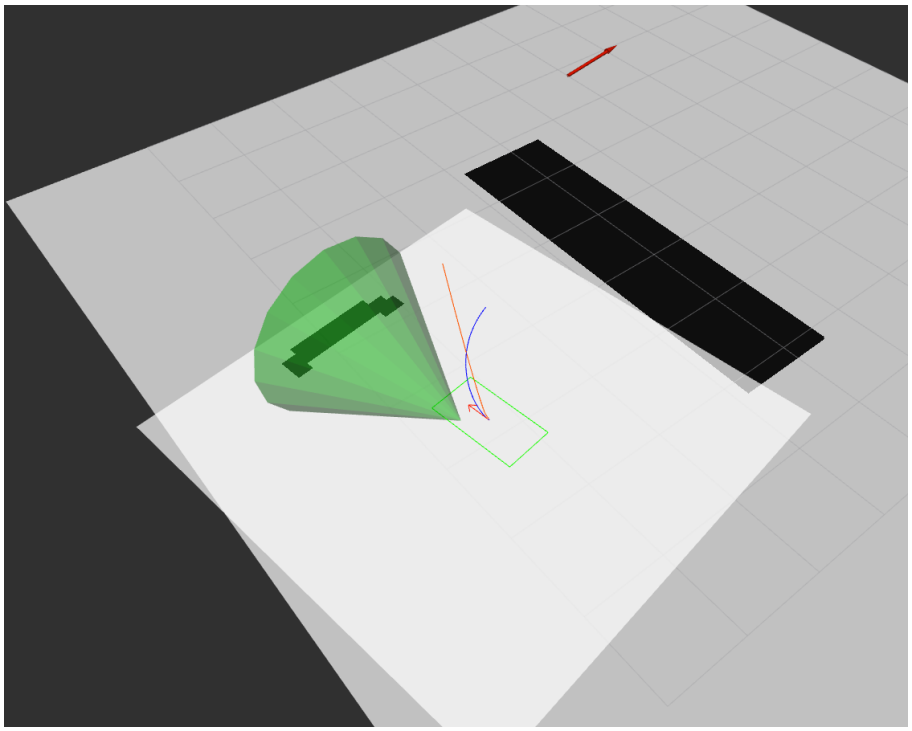


Figura 4.21: Obstáculo creado en base a la información de rango.

Para poder usar este *plugin* hay que crear las líneas del listado 4.24 en el fichero de configuración de *local_costmap*.

Listado 4.24: Declaración de range_sensor_layer

```
plugins:
   - {name: ultrasonic,   type: "range_sensor_layer::RangeSensorLayer"}
   - {name: inflation_layer, type: "costmap_2d::InflationLayer"}


 ultrasonic:
   clear_threshold:    0.9
   mark_threshold:     0.95
   no_readings_timeout: 2.0
   clear_on_max_reading: true
   topics: ["/SLIT_range", "/SLDD_range", "/STD_range", "/SDI_range","/           SLID_range
       ","/SJOY_range","/SLDT_range","/STI_range"]
```

Se puede apreciar que el sensor delantero derecho (SDD) no se va a usar para la generación de obstáculos, ya que se encuentra en el reposapiés de la silla y no ofrece información fiable. En su lugar se usa el sensor que se encuentra bajo el *Joystick* (SJOY).

4.2.5.5 Modificación de range_sensor_layer

En la implementación de este paquete se ha observado un problema. Cuando los sensores detectan un rango infinito, el paquete no limpia los obstáculos.

Este paquete solo limpia obstáculos cuando se envía un rango igual al máximo rango del sensor, pero no cuando no detecta nada (Rango infinito).

Para poder conseguir un correcto funcionamiento, se ha descargado el paquete desde el repositorio para poder editarlo y que se adapte al funcionamiento deseado en este trabajo.

Se ha editado el fichero *range_sensor_layer.cpp* añadiendo la condición de limpiar el obstáculo en caso de detectar el valor infinito (isinf) (listado 4.25).

Listado 4.25: Modificación del paquete range_sensor_layer

```
  if (range_message.range == range_message.max_range && clear_on_max_reading_)
    clear_sensor_cone = true;
  else if(isinf(range_message.range) && clear_on_max_reading_)
    clear_sensor_cone = true;
```

El paquete modificado se encuentra en el workspace de catkin, junto con el resto de paquetes creados en este trabajo.

4.2.6 Lanzamiento

Para poder lanzar todos los paquetes del *Navigation Stack* e incluir los ficheros de configuración, es necesario usar un fichero .launch.

A continuación se muestran las partes más importantes del código.

Listado 4.26: Lanzamiento de map_server con un mapa predefinido

```
<!-- Set the name of the map yaml file: can be overridden on the command line. -->
  <arg name="map" default="uah_map.yaml" />

  <!-- Run the map server with the desired map -->
```

```
<node name="map_server" pkg="map_server" type="map_server" args="$(find sara_2dnav)/maps/$(arg
     map)"/>
```

Listado 4.27: Agregar el fichero .launch de *move_base*

```
<!-- The move_base node -->
<include file="$(find sara_2dnav)/launch/fake/fake_move_base_amcl.launch" />
```

Al usar el comando *include* se agrega el código correspondiente al fichero apuntado. De esta forma el código queda más compacto y modular.

Listado 4.28: Lanzamiento de *move_base*

```
<node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen"
    clear_params="true">
```

En el fichero $fake_move_base_amcl.launch$ se encuentra el comando para lanzar el nodo *move_base* junto con la carga de ajustes, que no se ha incluido en esta parte de la memoria para mejorar la lectura.

Listado 4.29: Lanzamiento de fake_localization

```
 <node pkg="fake_localization" type="fake_localization" name="fake_localization" clear_params=
     "true" output="screen">
```

Lanzamiento de fake_localization junto con la carga de ajustes, que no se ha incluido en esta parte de la memoria para mejorar la lectura.

Listado 4.30: Agregar el fichero .launch de *Rviz*

```
<include file="$(find sara_2dnav)/launch/fake/fake_rviz.launch" />
```

En esta línea se agrega el fichero para lanzar la interface gráfica *Rviz*.

Listado 4.31: Lanzar el nodo *Rviz*

```
 <node pkg="rviz" type="rviz" name="fake_rviz"
args="-d $(find sara_2dnav)/rviz/fake_sensors_local_global.rviz"/>
```

Este es el código que aparece en el fichero incluido en la línea anterior. Lanza el nodo *Rviz* junto con el fichero de ajustes (*.rviz*), de forma que no se tenga que configurar la visualización cada vez que se lance.

Este fichero se puede generar desde los menús de la interface gráfica, guardando la configuración actual.

4.3 Transformación de sistemas de referencia de posición: frames

Una parte fundamental del funcionamiento de ROS se base en paquete tf [23], [24].

Este paquete crea una relación (offset) de traslación y rotación entre los distintos frames.

Por ejemplo, tenemos un robot con un sensor del cual sabemos la distancia que está midiendo *base_laser*, imagen 4.22.

Para poder navegar, necesitamos saber la distancia del obstáculo al centro del robot *base_link*. Este cálculo se podía realizar manualmente, pero ROS ofrece el paquete *tf*, que realiza este cálculo automáticamente.

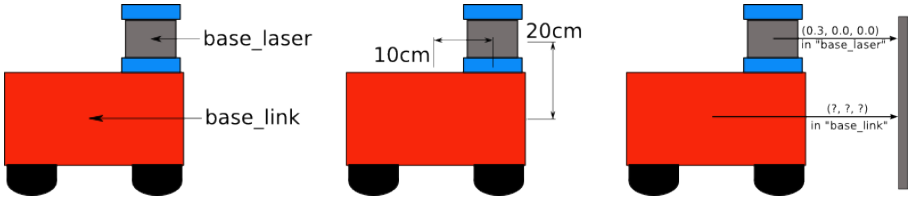


Figura 4.22: Ejemplo del uso de tf.

Definiendo la relación entre los dos frames *base_laser* y *base_link* , el paquete calculará de forma automática la distancia entre *base_link* y el obstáculo, imagen 4.23.

Es necesario elegir cuál será el padre y el hijo. En este caso la opción más optima es que el padre sea *base_link*, de forma que tenga la opción de tener varios hijos para los diferentes sensores.

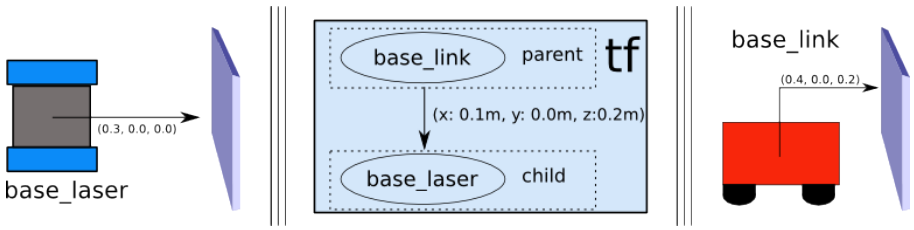


Figura 4.23: Ejemplo del uso de tf.

En el caso de este trabajo, el árbol de tf contiene múltiples frames. En la imagen 4.24 se muestra un recorte del arbol de frames que se han declarado.

- El frame *base_link* que se corresponde con la silla, tiene varios hijos por cada uno de los sensores de ultrasonidos. La relación entre *base_link* y los sensores es fija (los sensores no se mueven con respecto a la silla) y por lo tanto se puede usar el paquete *static_transform_publisher* [25] para establecer la relación, como se hace en el listado 4.32 .

Listado 4.32: Ejemplo de uso de *static_transform_publisher*

```
<node pkg="tf" type="static_transform_publisher" name="SLDTD_broadcaster" args="-0.17
    -0.21 0 -1.5708 0 0 base_link SLDT_frame 100" />
```

El paquete de Navegación usa dos frames adicionales: *odom* y *map*

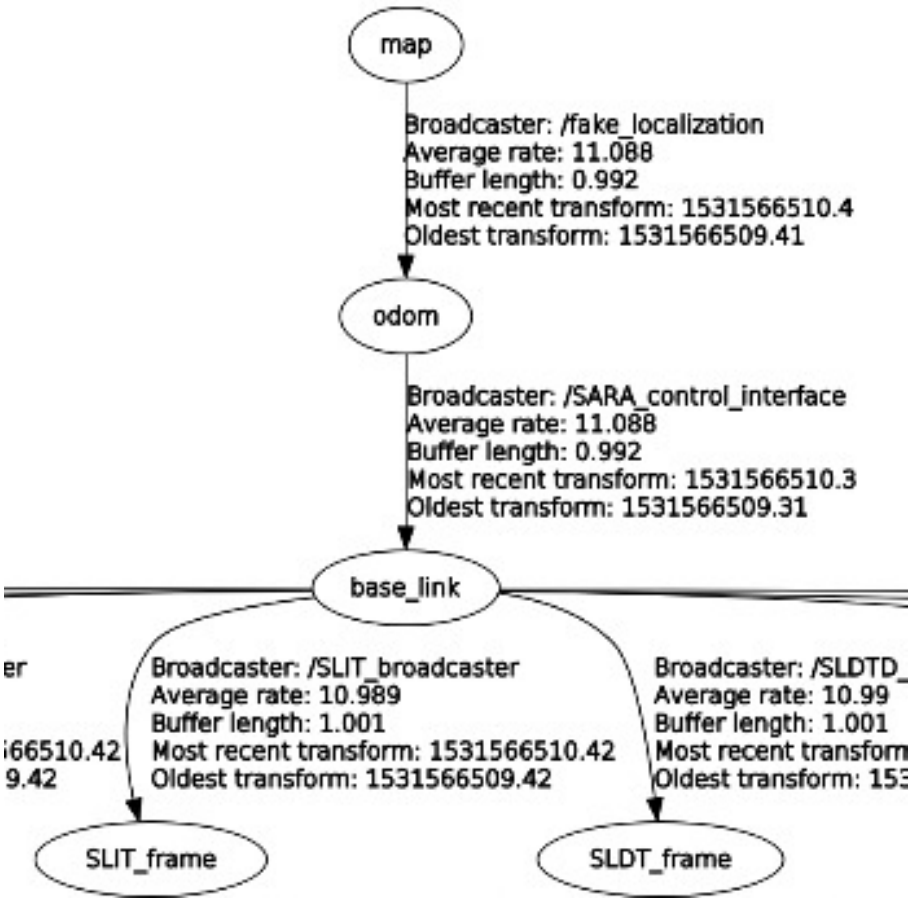


Figura 4.24: Árbol de *transform*.

- **odom** es el padre de base_link. Representa el mundo por el que se mueve la silla, desde el punto de vista de la odometría. El offset entre *odom* y *base_link* no es fija como pasaba con los sensores, si no que varía a medida que la silla se mueve por el mundo. La modificación del offset es generada y publicada por el controlador de alto nivel 4.1.3.

- **map** es el padre de odom. Representa el mundo que vemos en el mapa. Es el punto de referencia del resto de frames.

  En el offset entre *map* y *odom* se implementan las correcciones de navegación creadas por los sensores. Esto es debido a que la odometría produce errores debido al deslizamiento y las imperfecciones del modelado de las ruedas.

  Este offset es publicado por el paquete *AMCL* 4.2.4 gracias a la ayuda del sensor láser. Puesto que en este trabajo se ha sustituido el paquete *AMCL* por *fake_localizacion* debido a que no dispone de un sensor láser, el offset entre map y odom es nulo, ya que no se hace ninguna corrección.

4.4 Interface de usuario (*rviz*)

Ahora que ya se ha configurado el *stack*, es necesario un programa que permita interactuar con los usuarios, y donde poder introducir el destino al que se quiere llegar. El paquete *Rviz* [26], [27] permite visualizar de forma gráfica los *topics* generados por el *Navigation Stack* y de esa forma monitorizar el funcionamiento del sistema.

En la imagen 4.25 se muestra la *interface* que crea *rviz*, con la que se controla la silla.

- En el panel izquierdo se muestran todos los *topics* que se están mostrando en la representación. Estos *topic* se han seleccionado de forma manual y posteriormente se ha guardado la configuración. Durante el lanzamiento de *rviz* se restaura la configuración guardada usando el comando del listado 4.33.

Listado 4.33: Lanzamiento de *rviz* con ajustes preestablecidos.

```
<node pkg="rviz" type="rviz" name="fake_rviz"
  args="-d $(find sara_2dnav)/rviz/fake_sensors_local_global.rviz"/>
```

- El botón $2DPoseEstimate$ permite fijar la posición inicial del robot.
- El botón $2DNavGoal$ permite fijar el objetivo en el mapa donde la silla se moverá.

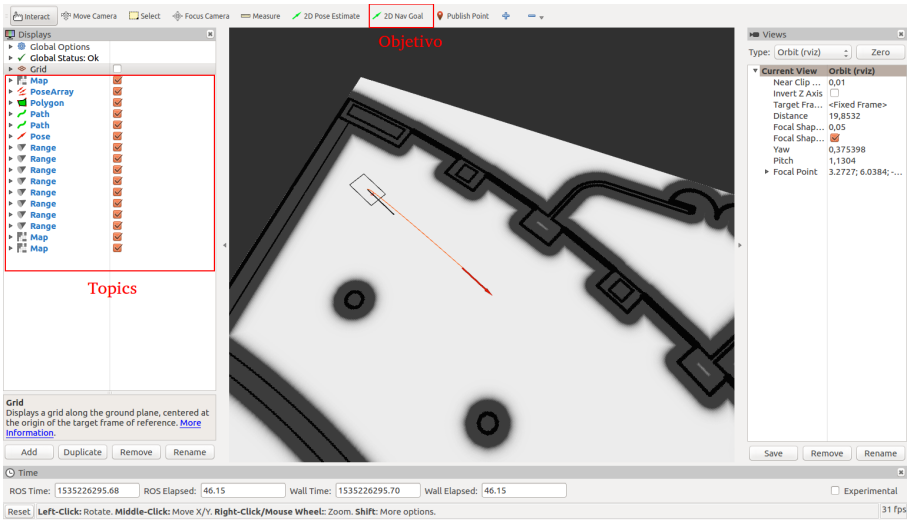


Figura 4.25: Visualizador *Rviz*.

También se puede ver en vídeo aquí https://www.youtube.com/watch?v=Ll5xFwT535o

4.5 Integración del modelo de la silla (URDF)

El modelo de la silla descrito en el apartado 4.5, se ha integrado con este trabajo, de forma que se visualice durante la representación de *rviz* como se muestra en la imagen 4.26.

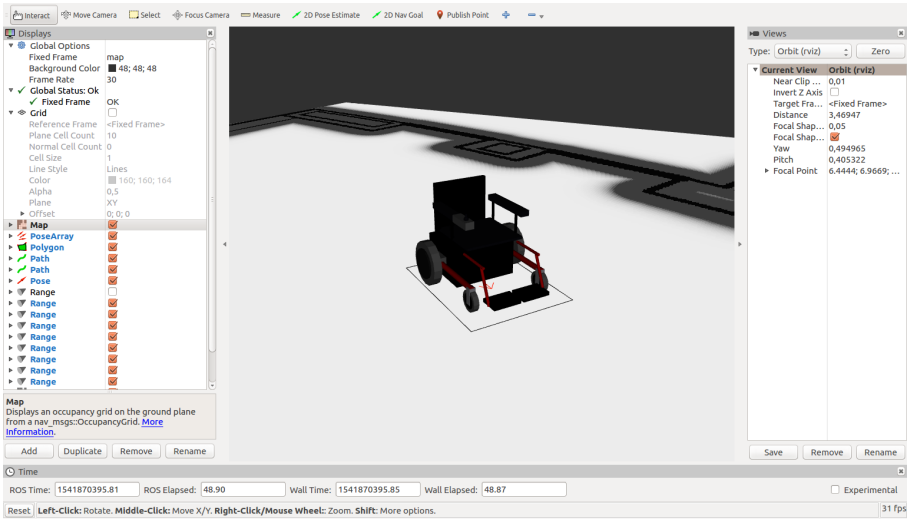


Figura 4.26: Visualización del modelo en *Rviz*.

Además de mejorar la presentación de *Rviz*, la incorporación del modelo ha permitido ajustar de forma precisa el *footprint*, usado para la evasión de obstáculos y ha permitido ajustar la posición de los sensores tal y como se encuentran montados en la silla, como se muestra en la imagen 4.27.

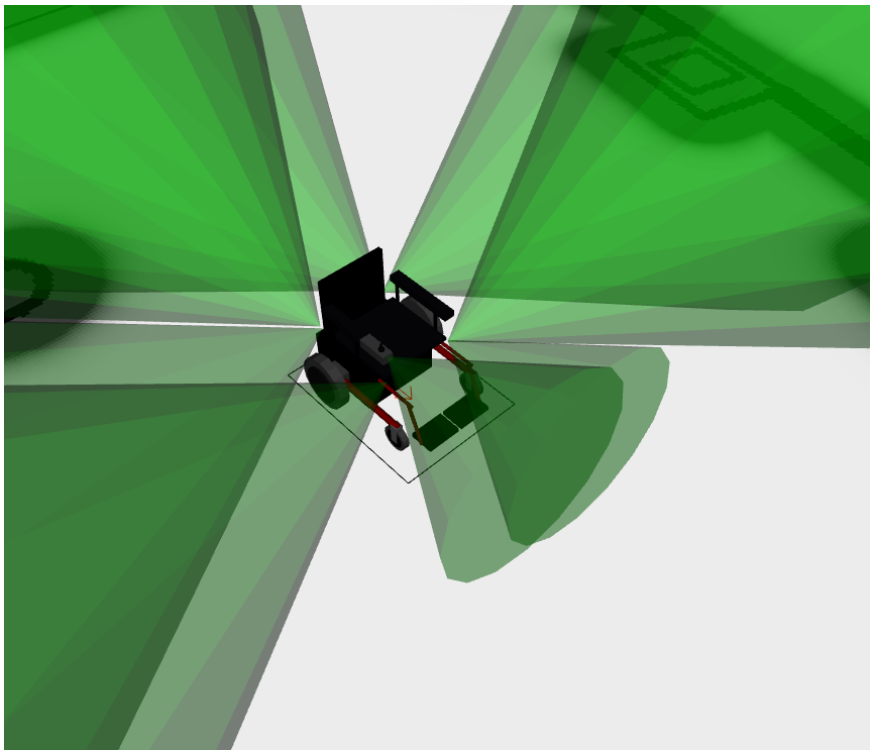


Figura 4.27: Ajuste de la posición de los sensores.

Para incluir el modelo en el trabajo se ha añadido en el paquete sara_model un carpeta con el modelo y se han incluido las lineas del listado 4.34 en el fichero sara_model_launch.launch.

Listado 4.34: Lanzamiento del modelo.

```
<arg name="model" default="$(find sara_model)/urdf/sara_urdf.xacro"/>
<param name="robot_description" command="$(find xacro)/xacro.py $(arg model)" />
<node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
```

Se puede ver un ejemplo del funcionamiento en el siguiente enlace: https://www.youtube.com/watch?v=lKqLOS70wqc

Capítulo 5

Resultados

5.1 Introducción

En este capítulo se introducirán los resultados más relevantes del trabajo. Se han creado varios escenarios para probar el sistema.

- Pruebas con la silla estática y consignas predefinidas.
- Pruebas con la silla estática, navegando en un entorno.
- Pruebas con la silla en movimiento y consignas predefinidas.
- Pruebas con la silla en movimiento, navegando en un entorno.

5.2 Grabación y análisis de los datos

Para analizar los *topics* que generan los diferentes nodos, y por lo tanto saber el estado del sistema, se ha usado el paquete de ROS rosbag [28].

Este paquete permite grabar un fichero tipo *.bag* con los *topics* deseados y si es necesario, volver a publicarlos posteriormente.

La grabación se puede realizar de dos formas:

- Llamando al nodo desde roslaunch.

Listado 5.1: Ejemplo de ejecución de rosbag

```
<node pkg="rosbag" type="record" name="rosbag_record_diag"  args=" -o /home//javier//
    bagfiles/vel_test.bag /cmd_wheel /cmd_vel /diff_drive_controller/odom /wheel_state /
    bat"/>
```

  En el fichero .launch se puede indicar la ruta del fichero y los *topics* que se quieren grabar.

- Usando el paquete rqt_bag: Esta utilidad permite realizar la grabación y reproducción de los *topic* mediante una interface gráfica.

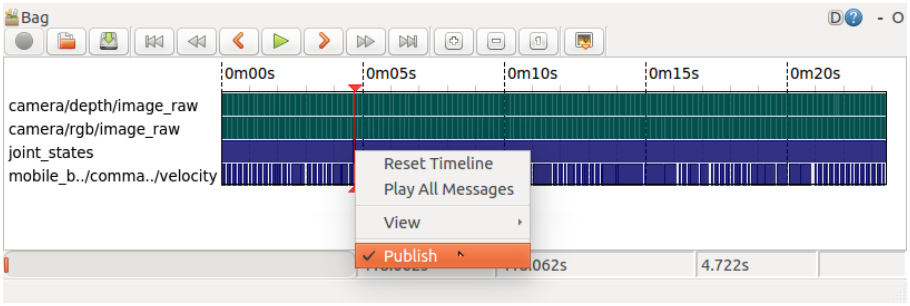


Figura 5.1: Interface rqt_bag.

No se ha usado esta opción pues es mucho más lento si se quieren hacer varias pruebas, pero para pruebas concretas en muy útil.

El análisis de los datos se ha realizado mediante MATLAB, ya que cuenta con un conjunto de herramientas para tratar con este tipo de ficheros. A continuación se muestran algunos ejemplos de cómo se debe trabajar con los ficheros de rosbag.

Listado 5.2: Importar un fichero de rosbag y seleccionar los *topic* deseados

```
bag = rosbag('estatico_corto_correccion.bag');

%%Velocidades angulares
cmd_wheel= select(bag,'Topic','/cmd_wheel');
wheel_state= select(bag,'Topic','/wheel_state');
```

Listado 5.3: Convertir los topics en timeseries para facilitar su anásis (No es posible hacerlo con todos)

```
odom_x = timeseries(odom,'Pose.Pose.Position.X');
odom_y =  timeseries(odom,'Pose.Pose.Position.Y');
```

Listado 5.4: Convertir los topic en una estructura de mensajes, su lectura es más tediosa que con timeseries, pero algunos topics sólo se pueden extraer de esta forma

```
cmd_wheel_struct = readMessages(cmd_wheel);
wheel_state_struct = readMessages(wheel_state);
```

5.3 Pruebas con la silla estática y consignas predefinidas

5.3.1 Estrategia y metodología de experimentación

Esta prueba se ha realizado con la silla en posición estática. Para ello se han levantado sus ruedas motrices de forma que giren libremente.

El objetivo de esta prueba ha sido ajustar el funcionamiento del controlador PID de bajo nivel y del control de alto nivel. Para ello se han introducido una serie de consignas de tipo escalón tanto de velocidad angular de las ruedas ω, como de velocidad de avance V y giro Ω de la silla de ruedas.

Para hacer esta prueba se han usado consignas mayores que los valores máximos configurados en el paquete de navegación, de esta forma se comprueba que no existe saturación en el conjunto controlador-planta.

5.3.2 Resultados experimentales

- La primera prueba (imágenes 5.2 y 5.3) se ha realizado sin ningún tipo de carga. Se puede apreciar que para consignas pequeñas existe un error muy pequeño de seguimiento de consigna en el régimen permanente, pero en rangos normales no se produce error.

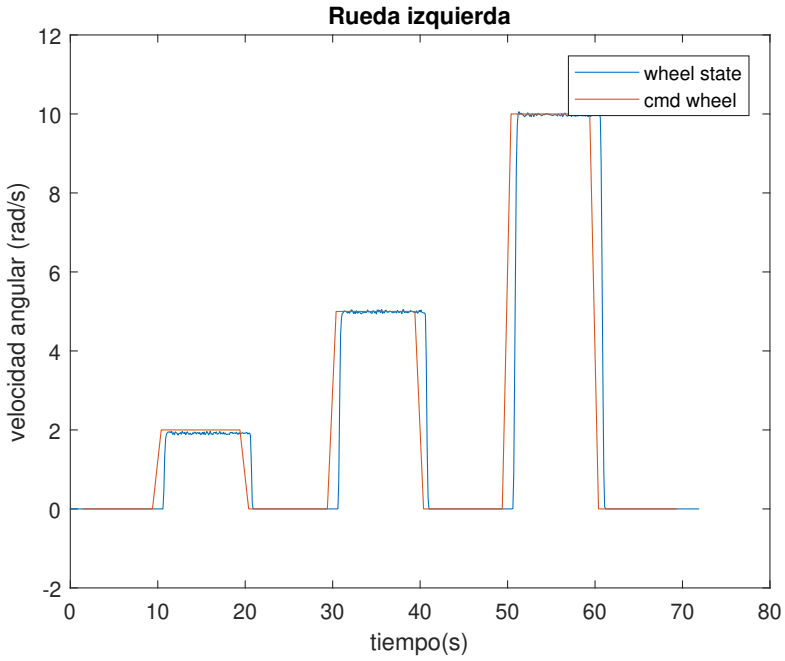


Figura 5.2: Respuesta de los motores ante varias consignas, sin perturbación.

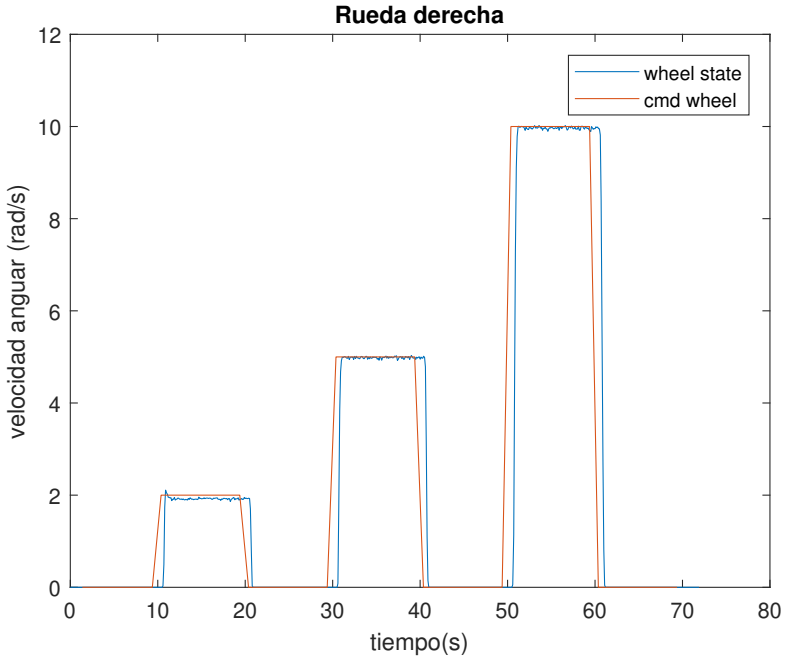


Figura 5.3: Respuesta de los motores ante varias consignas, sin perturbación.

- En la segunda prueba (imagen 5.4), se han introducido las mismas consignas pero creando una perturbación, al frenar manualmente las ruedas.

Se puede apreciar que el error de seguimiento es nulo en régimen permanente gracias al controlador PID integrado y no aparece un excesivo sobreimpulso en la salida ante la corrección de esta perturbación.

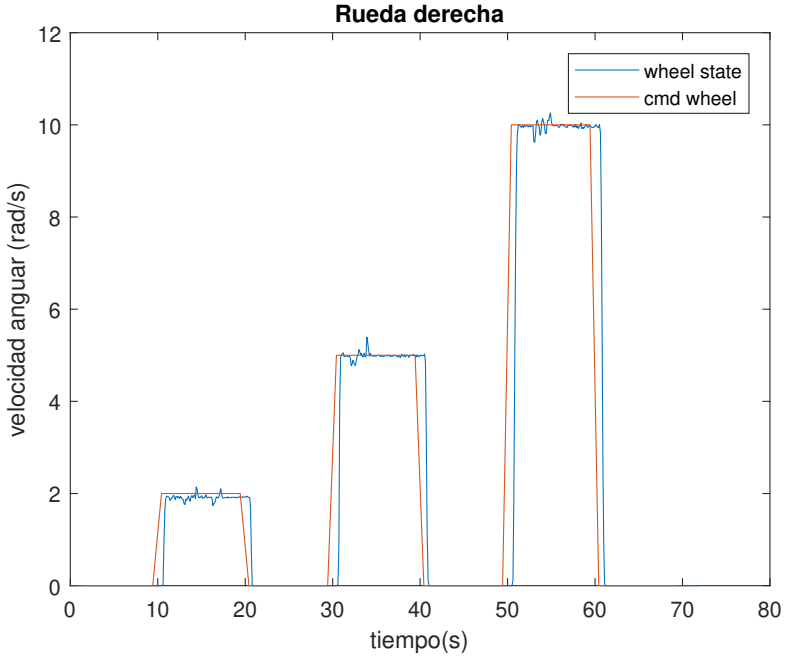


Figura 5.4: Respuesta de los motores ante varias consignas, con perturbación.

- Para terminar se ha analizado el retardo entre entrada y la salida de la planta (cada una de las ruedas con su controlador PI).

  En las imágenes 5.5 y 5.6 se han puesto cursores para medir el retado.

  Los retardos analizados varían entre $200ms$ y $300ms$ y puesto que el periodo de muestreo es de $100ms$ la planta tiene un retardo de entre $2Ts$ y $3Ts$

  Este retardo se produce principalmente en la omunicación entre el alto nivel y el bajo nivel, que se realiza mediante tramas CAN encapsuladas en un bus USB.

  Como se muestra en la imagen 5.7, la información debe pasar por múltiples etapas y numerosos cambios de formato, por lo que en cada una de ellas se produce cierto retardo.

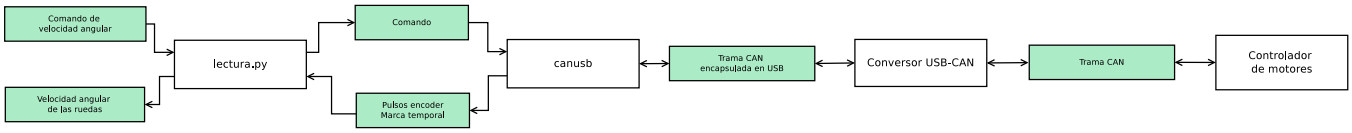


Figura 5.7: Etapas por las que pasan las consignas y la odometría.

A pesar de que existe retardo, no supone un gran problema, debido a que la planta es mucho más lenta.

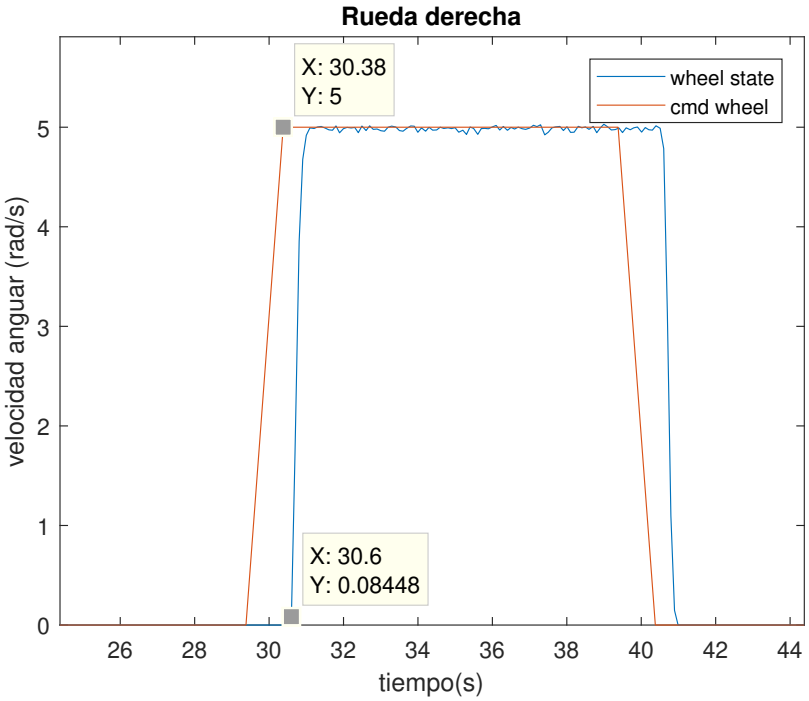


Figura 5.5: Retardo de la planta.

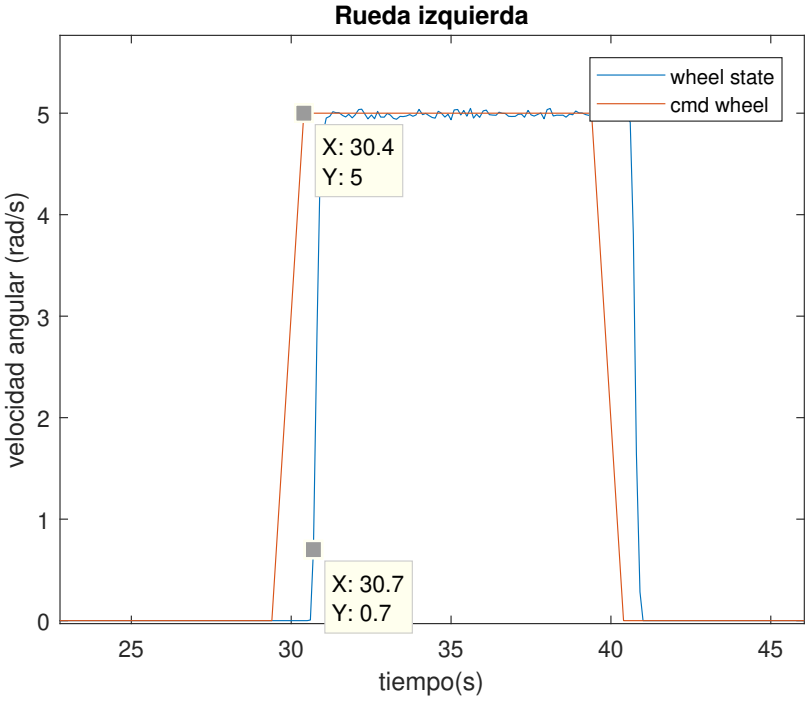


Figura 5.6: Retardo de la planta.

5.4 Pruebas con la silla estática, navegando en un entorno

5.4.1 Estrategia y metodología de experimentación

Esta prueba se realiza de forma estática al igual que en la prueba anterior, pero en este caso el test se ha realizado sobre el mapa real del entorno (imagen 4.14), quedando por lo tanto como una prueba de simulación.

Para indicar los puntos por los que debe pasar la silla de ruedas autónoma se ha usado el paquete *SimpleActionClient* [3], que permite fijar una cadena de objetivos que el sistema robótico tiene que alcanzar, quedándose en espera mientras se alcanzan cada una de las metas.

En las imágenes 5.8, 5.9, 5.10, 5.11 y 5.12 se muestran las metas introducidas con el paquete SimpleActionClient y las rutas creadas por el *global_planner*. La información mostrada por los sensores de ultrasonido no es correcta pues al ser una prueba con la silla estática los valores no son reales.

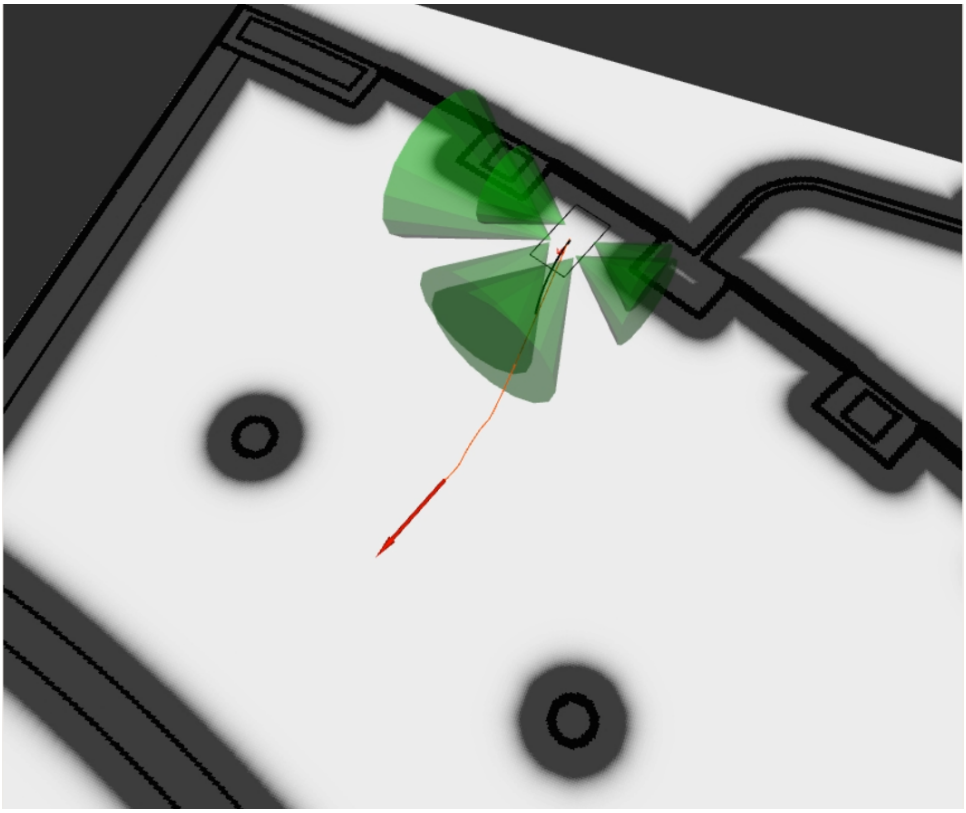


Figura 5.8: Recorrido simulado en el mapa de la terraza, con la silla de ruedas estática.

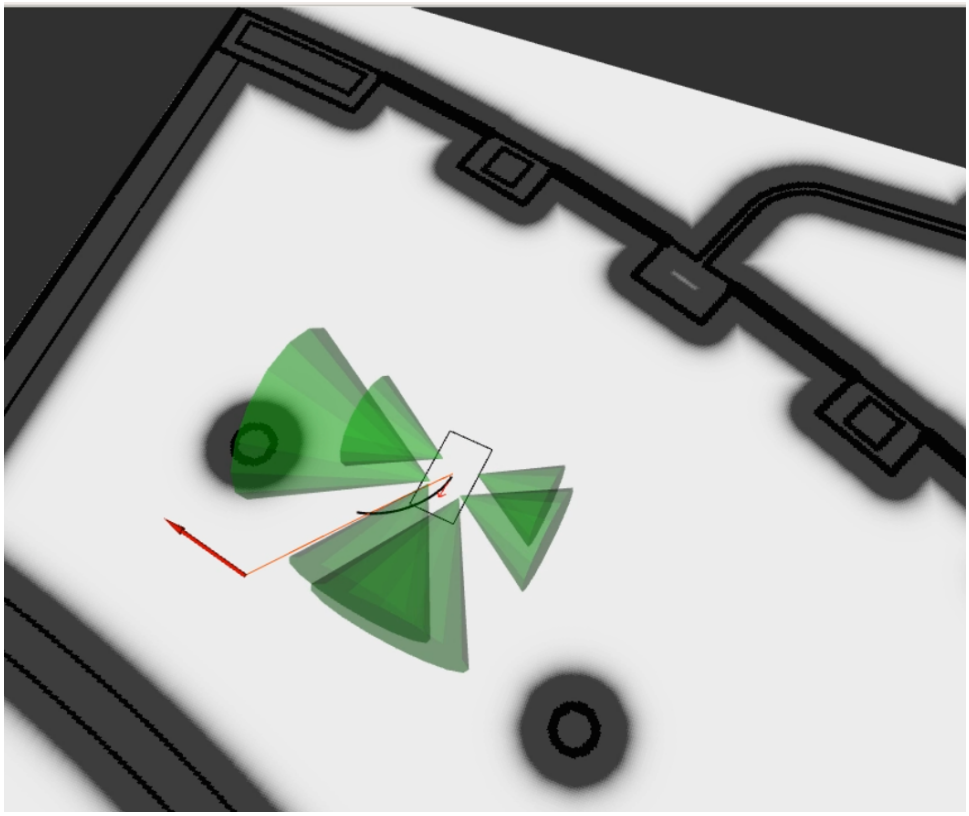


Figura 5.9: Recorrido simulado en el mapa de la terraza, con la silla de ruedas estática.

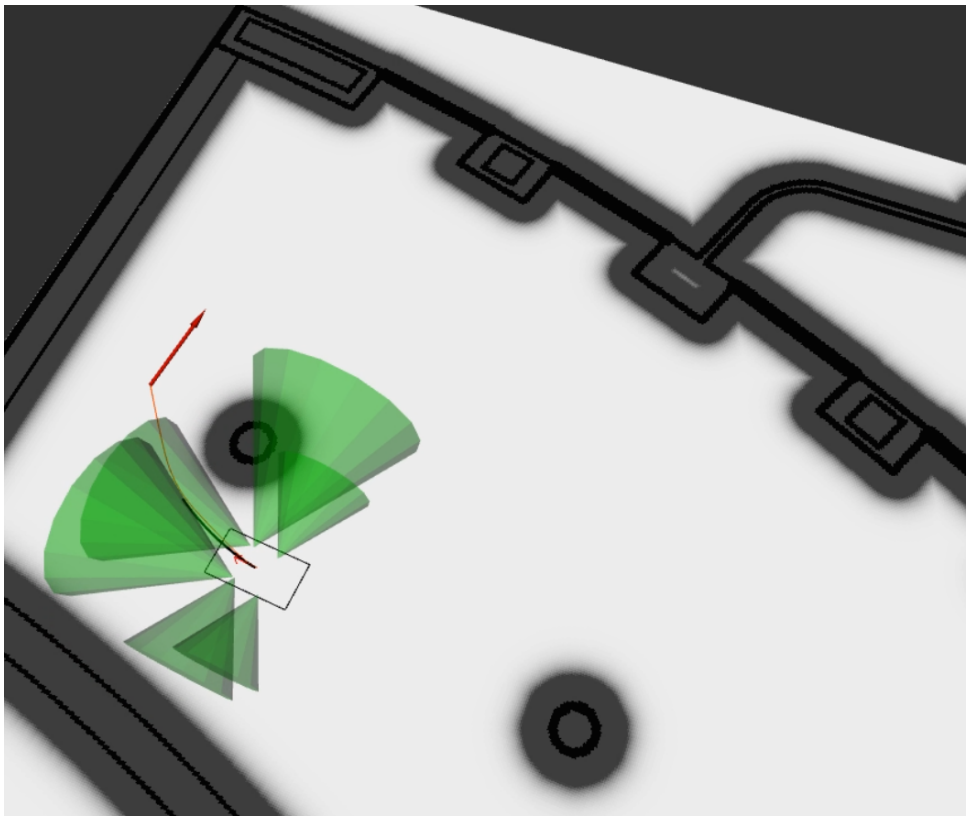


Figura 5.10: Recorrido simulado en el mapa de la terraza, con la silla de ruedas estática.

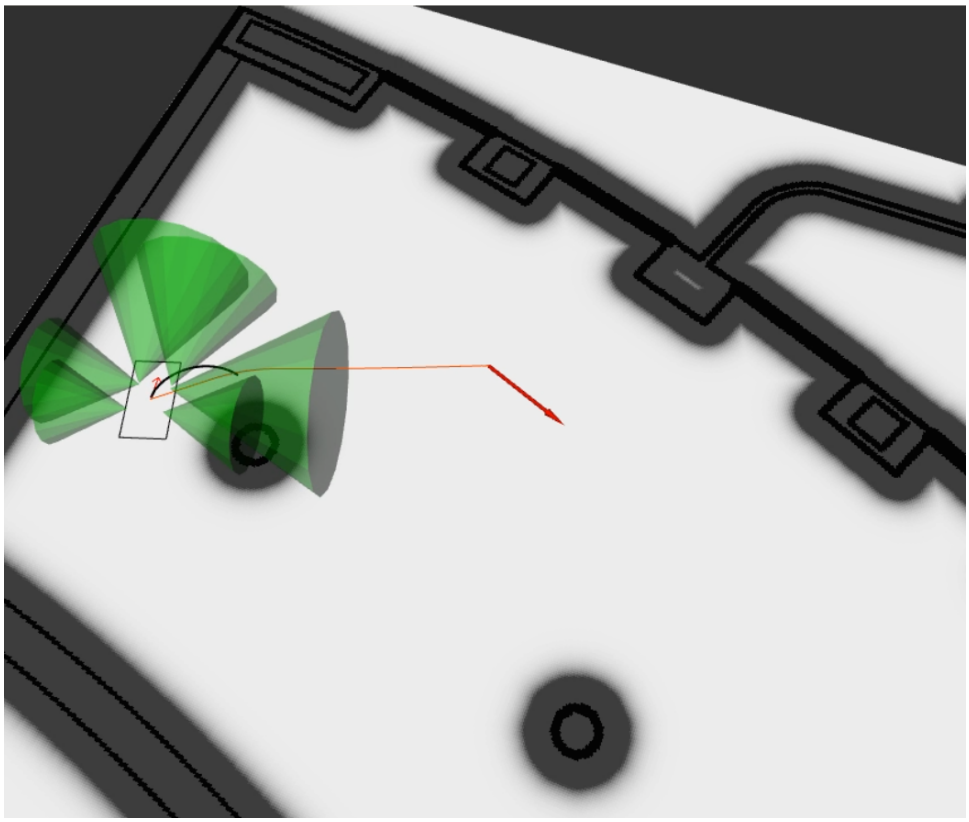


Figura 5.11: Recorrido simulado en el mapa de la terraza, con la silla de ruedas estática.

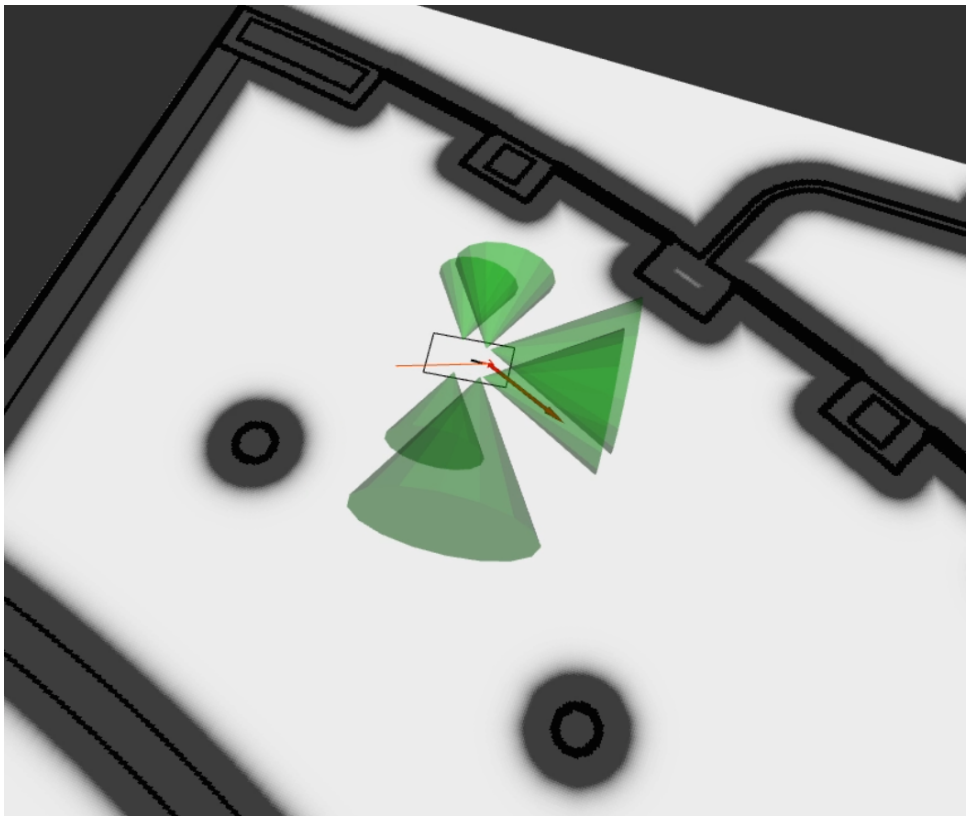


Figura 5.12: Recorrido simulado en el mapa de la terraza, con la silla de ruedas estática.

5.4.2 Resultados experimentales

En esta prueba, el sistema de navegación crea las consignas de velocidad de avance y velocidad de giro que son enviadas al controlador de alto nivel, que a su vez genera las consignas de velocidad angular para las ruedas de la silla. A continuación se va a mostrar un ejemplo de las consignas mencionadas en el párrafo anterior, junto con la realimentación de cada una de ellas.

- En primer lugar se van a analizar las consignas de velocidad angular para las ruedas que se muestran en las imágenes 5.13 y 5.14.

  Se puede observar que el controlador *PID* de bajo nivel consigue seguir las consignas con un error reducido y sin que aparezca ningún sobreimpulso excesivo en la salida.

  También se aprecia que las consignas generadas por el control de alto nivel, no son de tipo escalón, gracias a los ajustes de aceleración que se describen en el apartado 4.1.3.2. Gracias a esta característica, se reducen en gran medida los sobreimpulsos del PID y se reducen los errores en régimen transitorio.

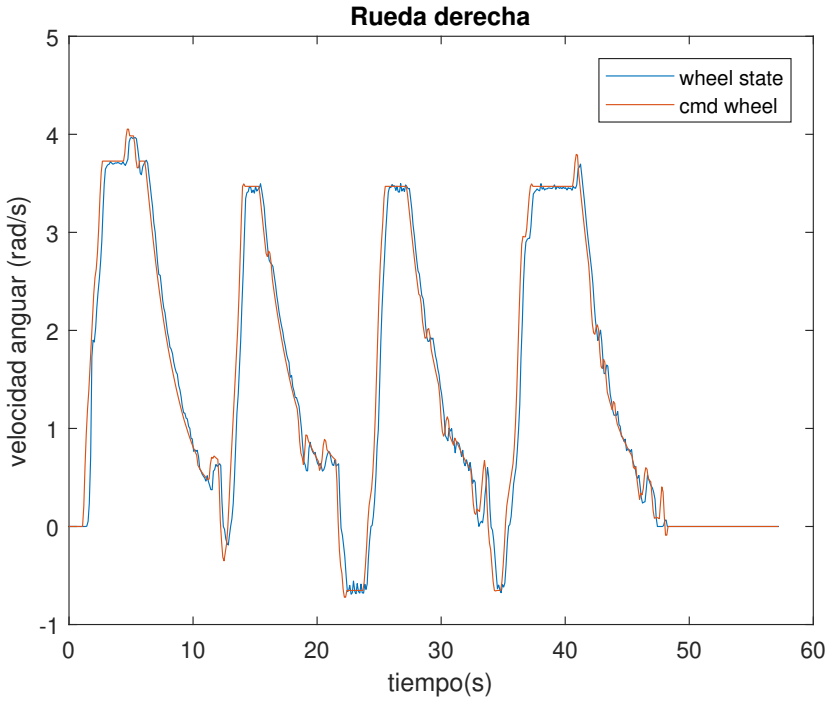


Figura 5.13: Velocidad angular de las rueda derecha.

- El retardo sigue siendo reducido en esta prueba, como se muestra en la imagen 5.15.

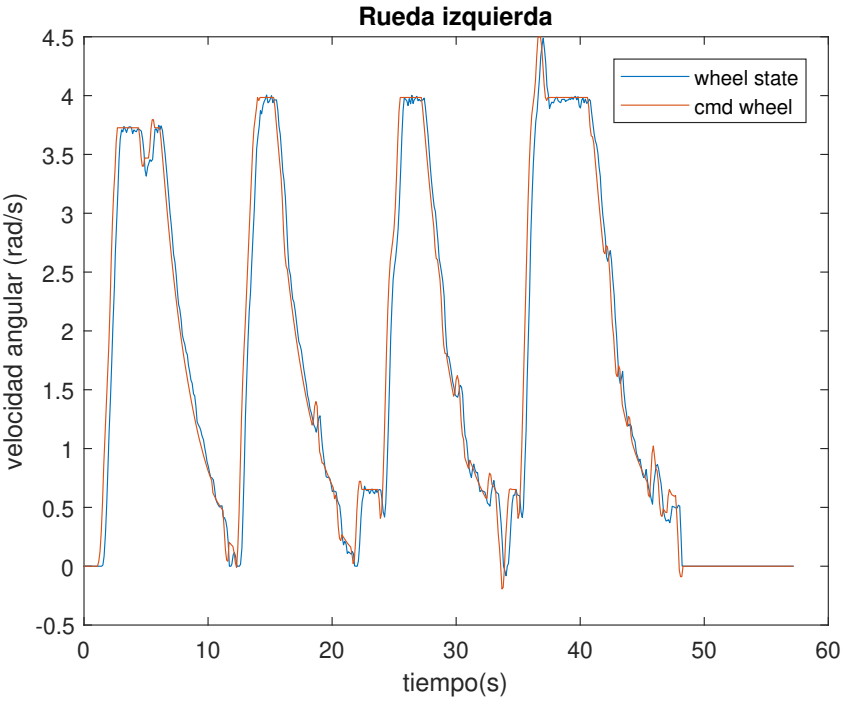


Figura 5.14: Velocidad angular de las rueda izquierda.

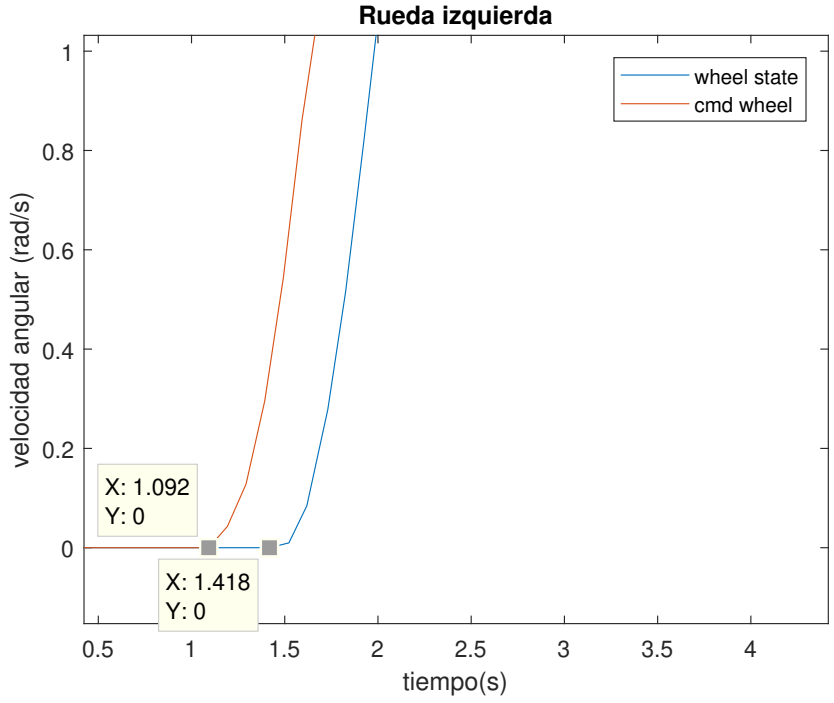


Figura 5.15: Retardo en la velocidad angular de las ruedas.

- A continuación se analizan las consignas de velocidad de avance (imagen 5.16) y velocidad de giro de la silla de ruedas (imagen 5.17) generadas por el sistema de navegación.

  La gráfica de la velocidad de avance de la silla de ruedas 5.16, muestra que las consignas generadas por el sistema de navegación son de tipo escalón y saturadas a un valor de $0{,}6m/s$, este valor se ajusta en las opciones del sistema de navegación (paquete *local_planner*), en el capítulo 4.2.2.4.

  Por otro lado, cuando es necesario parar la silla, se realiza de forma progresiva, hasta que se alcanza una velocidad baja $0{,}1m/s$, donde se produce una bajada brusca hasta $0m/s$ para no pasar por la zona muerta de los motores y evitar oscilaciones.

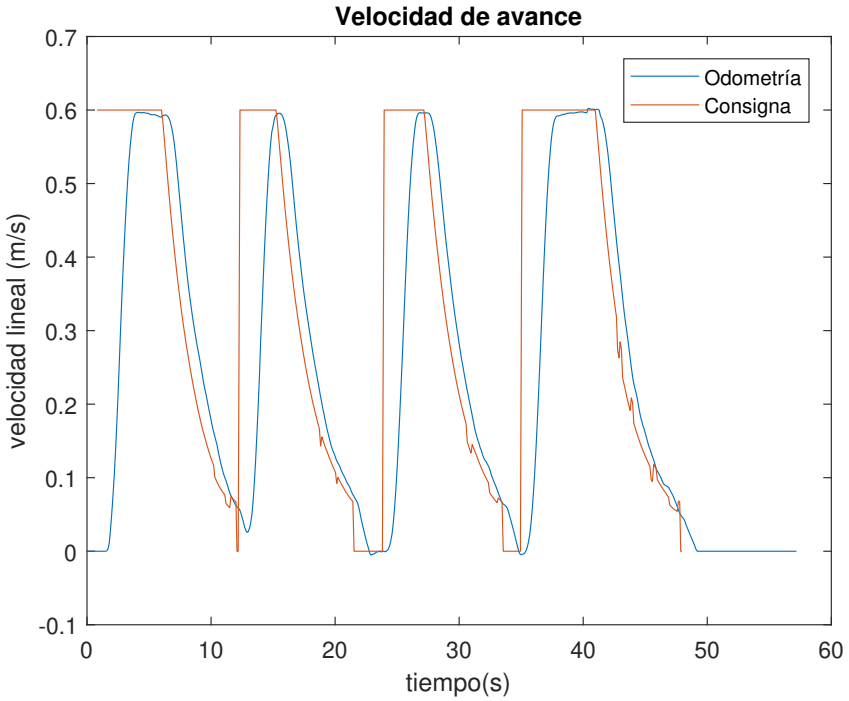


Figura 5.16: Velocidad de avance.

En cuanto a la gráfica de la velocidad de giro de la silla de ruedas 5.17, se puede apreciar que las consignas están cuantizadas. Es decir se producen saltos discretos con un valor mínimo de $0{,}15rad/s$.

- Se aprecia un ligero incremento en el retardo debido a que existe una etapa más en el lazo de control, el controlador de alto nivel.

  En la imágenes 5.18 y 5.19 se muestra que el retardo es de unos $400ms$, es decir $4T_s$.

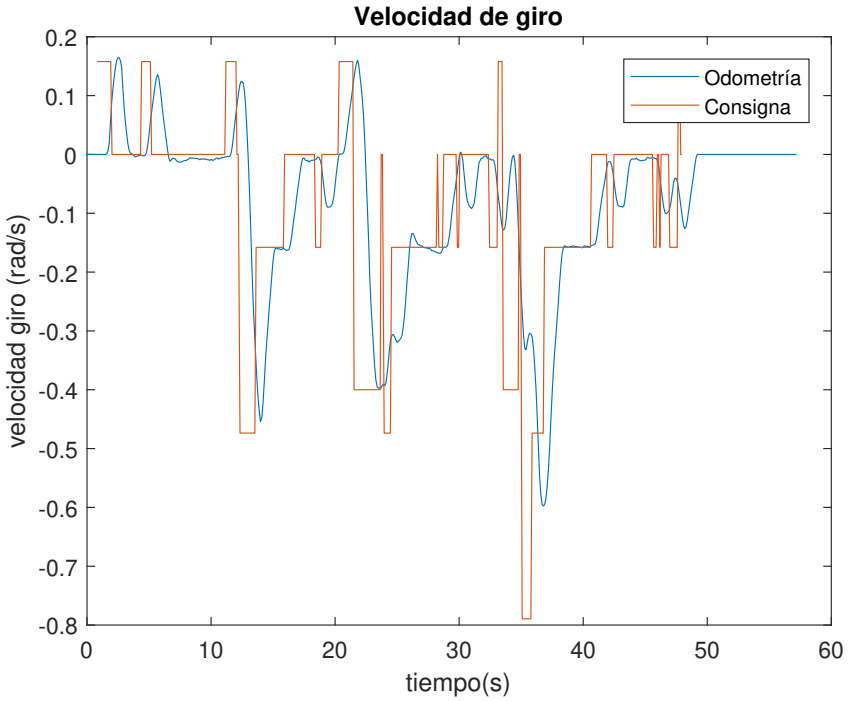


Figura 5.17: Velocidad de giro.

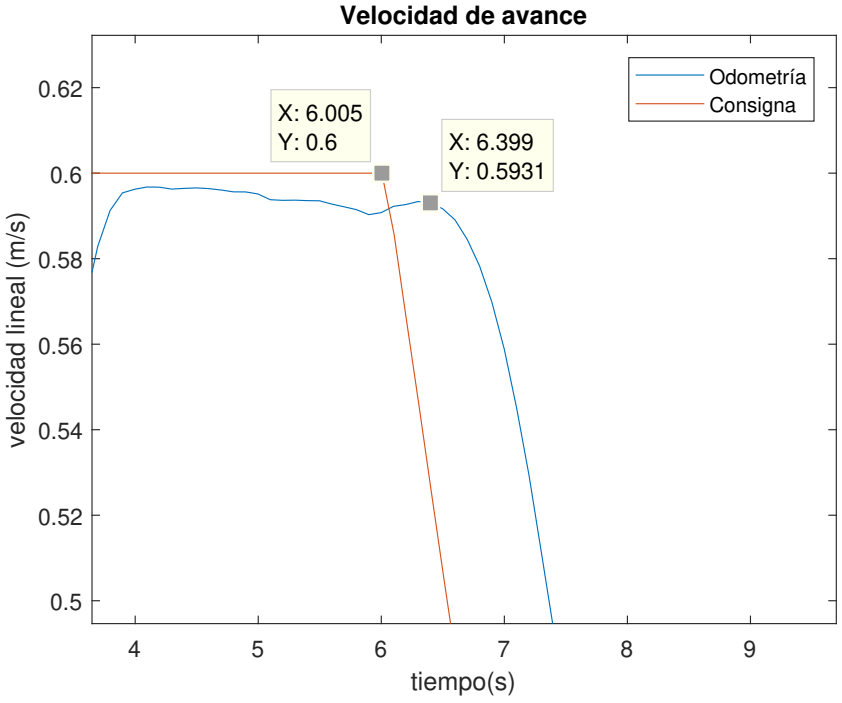


Figura 5.18: Retardo de la velocidad de avance.

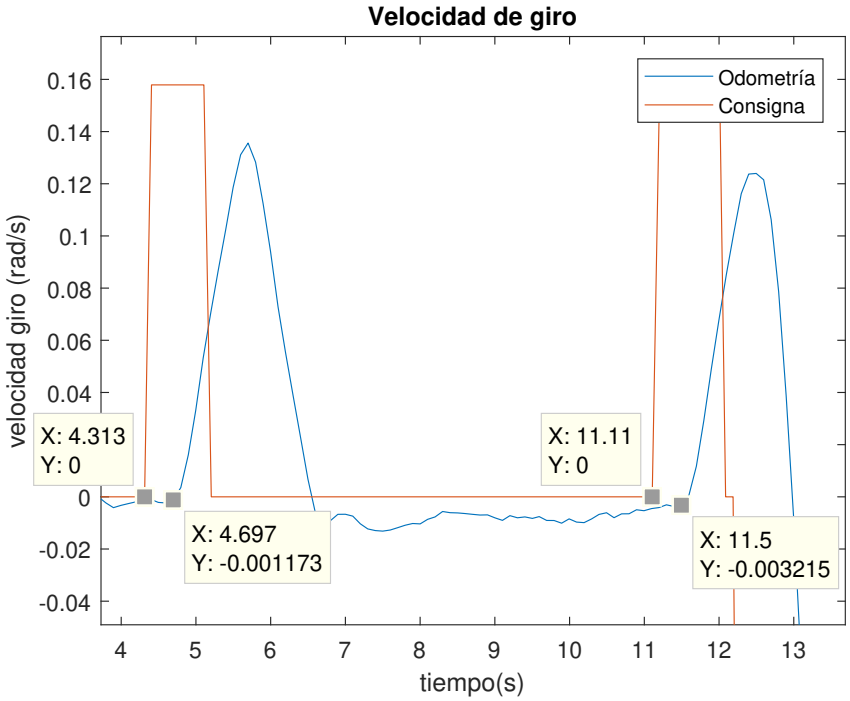


Figura 5.19: Retardo de la velocidad de giro.

5.5 Pruebas con la silla en movimiento y consignas predefinidas

5.5.1 Estrategia y metodología de experimentación

La última etapa de las pruebas se ha realizado con la silla de ruedas en movimiento, con una persona sobre ella.

En primer lugar se ha realizado una prueba de avance en linea recta, fijando las consignas manualmente, que permite analizar de forma más sencilla el comportamiento del sistema de control de bajo y alto nivel.

A diferencia del apartado 5.3, la consigna introducida ha sido de velocidad de avance en lugar de velocidad angular de las ruedas, por lo que en esta prueba también entra en juego el controlador de alto nivel.

5.5.2 Resultados experimentales

- La consigna introducida para esta prueba es de $0{,}6 m/s$ (imagen 5.20), que se corresponde con la velocidad máxima fijada para el sistema de navegación, por lo que se ajusta al funcionamiento real, cuando se use el sistema de navegación.

  Puesto que ha aumentado la carga, el comportamiento dinámico de la silla cambia y el tiempo de subida debería de haber aumentado con respecto al apartado 5.6. Esto no es así ya que al fijar una aceleración máxima en el controlador de alto nivel, se limita en ambos casos el tiempo de subida. Es decir el comportamiento dinámico está completamente definido por los parámetros del controlador de alto nivel.

  En esta imagen también se muestra que el retardo no cambia, ya que no depende de la carga de la silla.

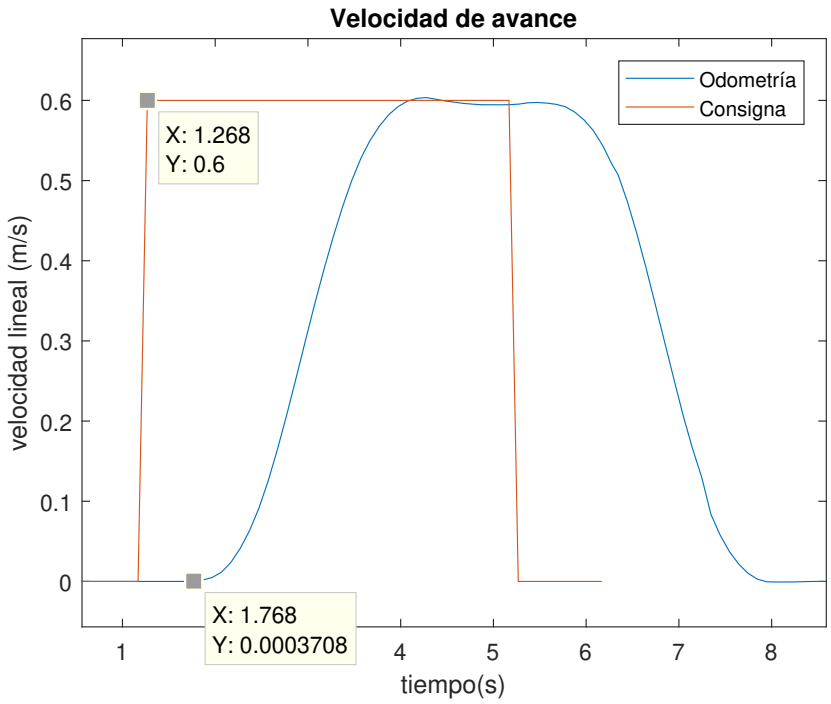


Figura 5.20: Velocidad de avance.

- En la imagen 5.21 se muestran las consignas de velocidad angular enviadas a una de las ruedas. La pendiente de las consignas se produce debido a la limitación de aceleración del controlador de alto nivel. La silla no tiene problemas en seguir las consignas a pesar de tener carga.

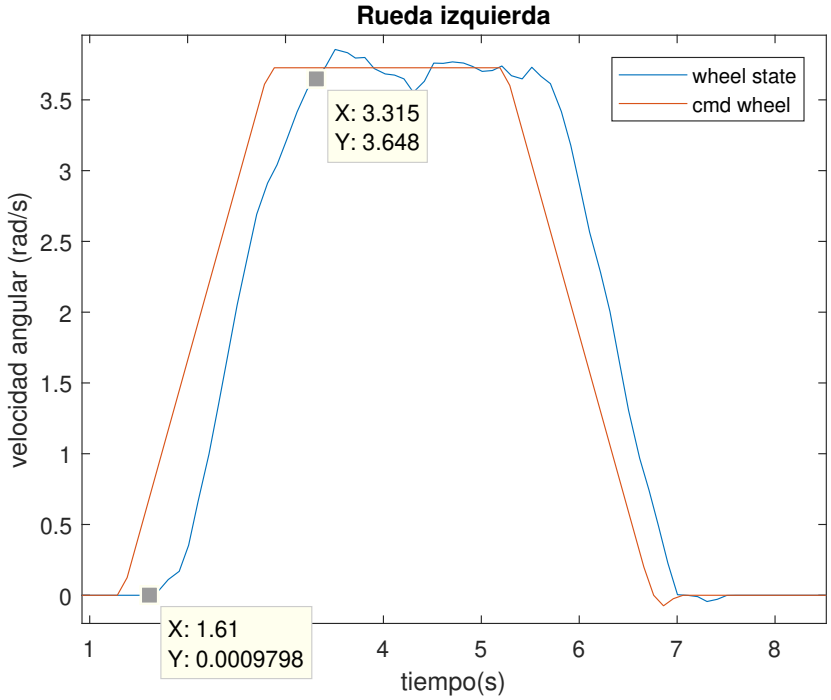


Figura 5.21: Velocidad angular de la rueda izquierda con el tiempo de subida.

- En esta prueba se ha experimentado el problema de no contar con un sistema de posicionamiento absoluto como un sensor láser.

  Debido a que una de las ruedas estaba ligeramente más inflada que la otra, la silla se desvía de la trayectoria deseada, sin que el navegador tenga conocimiento de ello, pues la odometría no es capaz de medir este error, que además es acumulativo.

  Para intentar minimizar el error se ha usado uno de los parámetros del controlador de alto nivel (*diff_drive_controller*), que permite ajustar un tamaño diferente para cada una de las ruedas. Gracias a este parámetro el controlador tiene en cuenta la diferencia de diámetro y envía consignas modificadas a las ruedas para compensar el error.

  Los parámetros son $left\_wheel\_radius\_multiplier$ y $right\_wheel\_radius\_multiplier$. Estos parámetros no aparecen en la documentación pero sí en el código fuente como se muestra en el listado 5.5.

Listado 5.5: Adquisición de los parámetros $left\_wheel\_radius\_multiplier$ y $right\_wheel\_radius\_multiplier$

```
if (controller_nh.hasParam("wheel_radius_multiplier"))
{
  double wheel_radius_multiplier;
  controller_nh.getParam("wheel_radius_multiplier", wheel_radius_multiplier);

  left_wheel_radius_multiplier_  = wheel_radius_multiplier;
  right_wheel_radius_multiplier_ = wheel_radius_multiplier;
}
```

```
    else
    {
      controller_nh.param("left_wheel_radius_multiplier", left_wheel_radius_multiplier_,
          left_wheel_radius_multiplier_);
      controller_nh.param("right_wheel_radius_multiplier", right_wheel_radius_multiplier_
          , right_wheel_radius_multiplier_);
}
```

Este método minimiza el error, pero nunca lo corrige definitivamente, puesto que la diferencia de diámetro va a depender de la persona que use la silla de ruedas autónoma, el estado de las ruedas y el tipo de superficie por la que se circule.

Por lo tanto queda de manifiesto que es imprescindible contar con un sistema posicionamiento de absoluto, para lograr una navegación precisa.

5.6 Pruebas con la silla en movimiento, navegando en un entorno

5.6.1 Estrategia y metodología de experimentación

Las pruebas finalizan en este apartado en el que se analizan las condiciones de funcionamiento finales del sistema de navegación. Un usuario usando la silla de ruedas manejando el sistema de navegación autónoma.

En esta prueba se ha fijado un destino, siendo el sistema de navegación el que genera todas las consignas.

5.6.2 Resultados experimentales

- En primer lugar se van a analizar las consignas de velocidad angular para las ruedas, que se muestran en las imágenes 5.22 y 5.23.

  En estas imágenes se puede apreciar que el PID produce un sobreimpulso mayor que en apartado anterior.

  Esta situación se debe a la forma en la que se ha iniciado la prueba:

  – En primer lugar se ha iniciado el sistema de navegación.
  – A continuación se ha fijado la meta.
  – Para terminar se ha pulsado el botón 3 del *joystick*, que habilita el envío de consignas al bus CAN.

  Esta situación provoca que la consigna enviada al PID sea un escalón y no una rampa con una aceleración controlada. Para eliminar este problema es conveniente pulsar el botón 3 del *joystick* antes de fijar una meta.

- A continuación se analiza el comportamiento de la velocidad de avance y giro de la silla de ruedas (imágenes 5.24 5.25)

  En estas imágenes se puede apreciar un sopreimpulso inicial en la velocidad de avance debido al problema comentado en el párrafo anterior.

  El comportamiento de las consignas se ha explicado ya en el apartado 5.4.2.

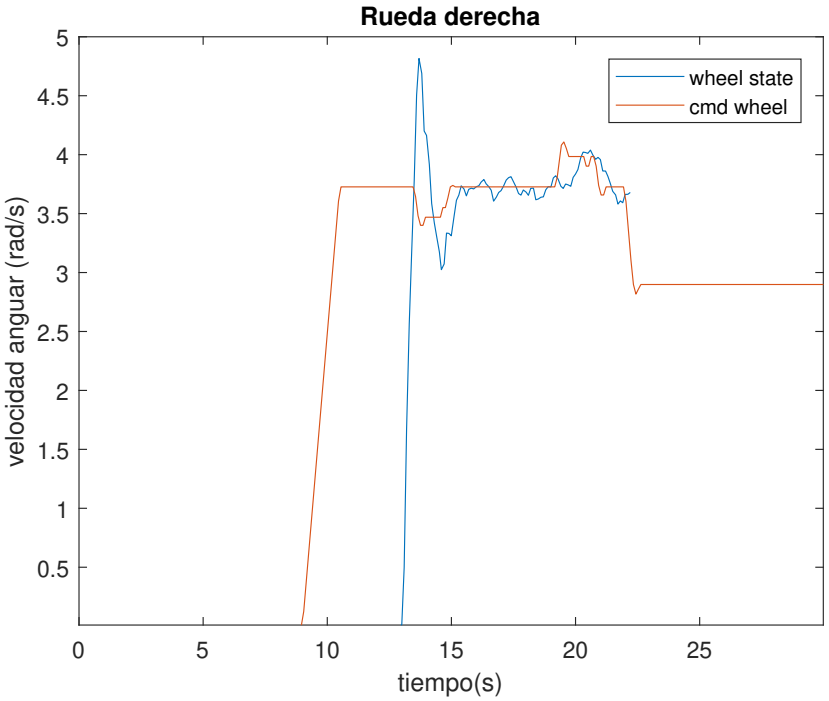


Figura 5.22: Velocidad angular de la rueda derecha.

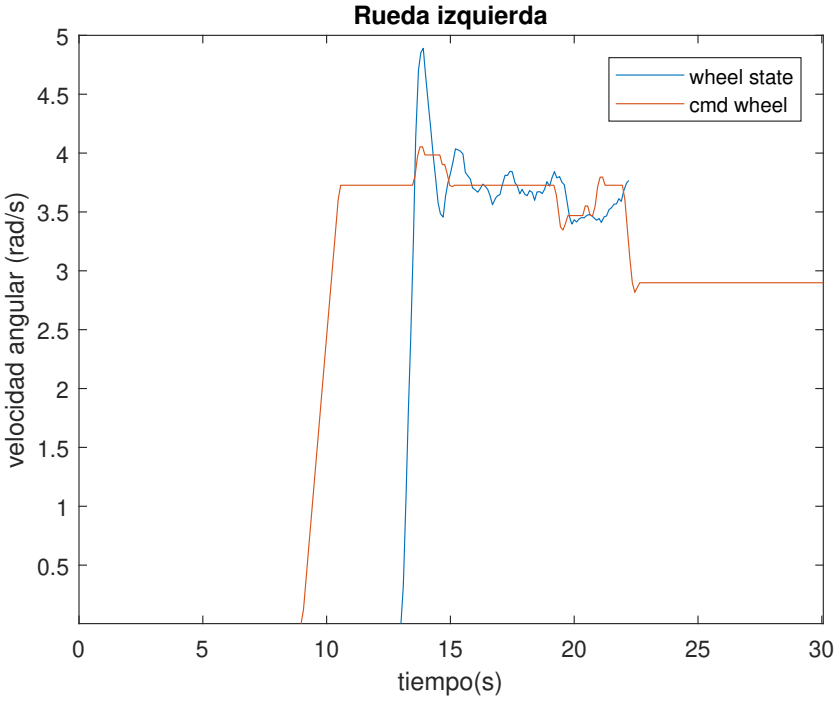


Figura 5.23: Velocidad angular de la rueda izquierda.

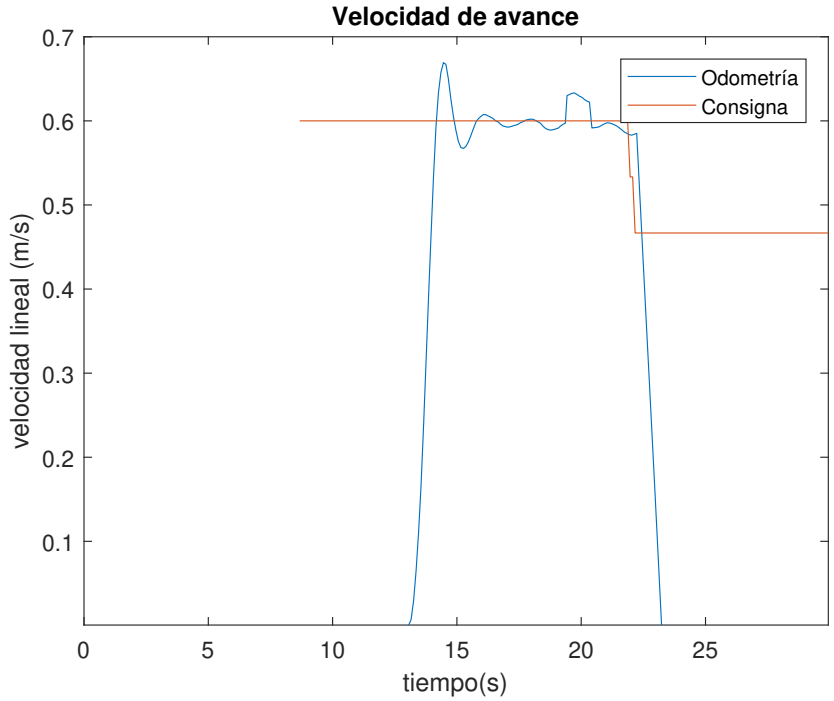


Figura 5.24: Velocidad de avance de la silla de ruedas.

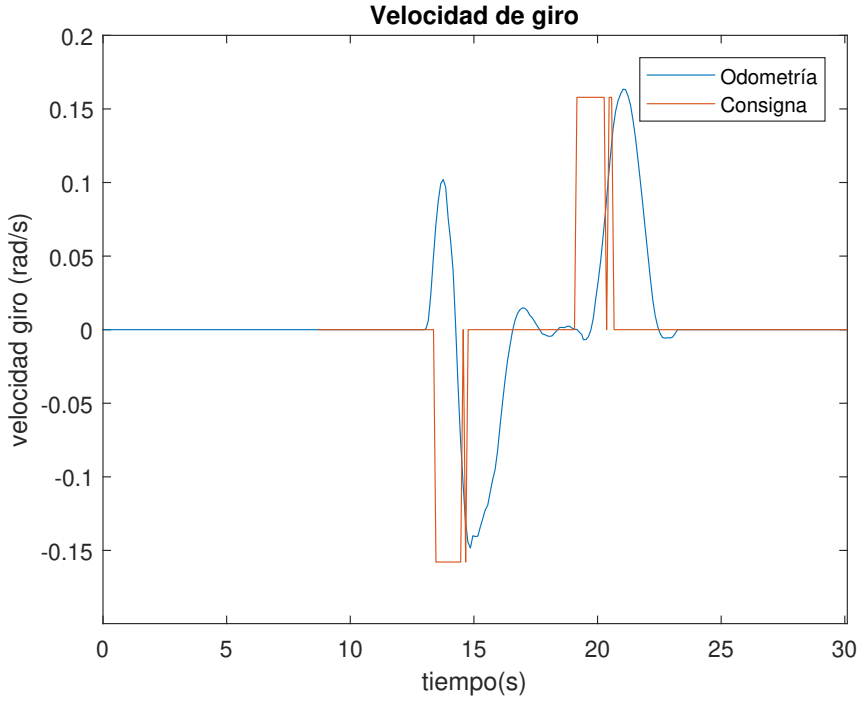


Figura 5.25: Velocidad de giro de la silla de ruedas.

5.7 Conclusiones

A lo largo de las sucesivas pruebas realizadas se ha verificado el correcto funcionamiento del control y la navegación.

- El control PID de bajo nivel funciona correctamente, con error nulo de seguimiento en régimen permanente, con poco sobreimpulso y un retardo de $2 - 3T_s$.
- El control de alto nivel genera correctamente las consignas de velocidad angular, con la pendiente necesaria para limitar la aceleración. El seguimiento es preciso, aunque aumenta ligeramente el retardo.
- El sistema de navegación crea las rutas de forma correcta, evadiendo los obstáculos del mapa y los detectados por los sensores de utrasonidos.

  Sin embargo este sistema de navegación está basado en la odometría, que permite estimar,no determinar su posición relativa con respecto al la posición inicial.

  La odometría es sencilla y barata puesto que solo es necesario el uso de encoders en las ruedas, pero produce una acumulación de errores a medida que se aumenta la distancia recorrida, debido al deslizamiento de las ruedas y la diferencia de diámetro de estas en función del peso de usuario de la silla.
- En conclusión, el sistema de navegación basado en la odometría es una buena base, pero debe de ser complementada por sistemas de posicionamiento absoluto como sensores láser, que proporcionan una estimación de la posición más fiable.

5.8 Problemas

Durante la realización del trabajo se han encontrado errores en algunos paquetes de ROS. Algunos han podido ser solucionados, pero otros a fecha de entrega de este trabajo, están pendientes de resolver por la comunidad de desarrolladores. Estos errores han sido reportados por los canales oficiales, confirmando el problema.

- Problema al fijar la posición inicial de la silla de ruedas durante la ejecución del sistema de navegación: Este problema se localiza en el paquete *fake_localization*. Ha sido reportado en el siguiente hilo: https://github.com/ros-planning/navigation/issues/796.
- Problema con el parámetro *delta_yaw* del paquete *fake_localization*, que permite fijar el ángulo en el que el robot aparece en mapa, al arrancar el sistema de navegación. Ha sido reportado en el siguiente hilo: https://github.com/ros-planning/navigation/issues/805.
- Problema en el controlador *diff_drive_controller*. Al iniciarse aparece el mensaje de la imagen 5.26.

  Según se muestra en la imagen, parece que han cambiado los valores del radio y la separación, pero los valores mostrados son los multiplicadores de ambos parámetros, como aparece en la imagen 5.27.

  Ha sido reportado en el siguiente hilo https://github.com/ros-controls/ros_control/issues/356.
- Problema con el paquete *range_sensor_layer*. Los obstáculos no se limpian correctamente. El problema ha sido solucionado en el apartado 4.2.5.5.

```
[ WARN] [1537115999.720664105]: updateConfig() called on a dynamic_reconfigure::
Server that provides its own mutex. This can lead to deadlocks if updateConfig()
 is called during an update. Providing a mutex to the constructor is highly reco
mmended in this case. Please forward this message to the node author.
[ INFO] [1537115999.730753809]: Dynamic Reconfigure:
DynamicParams:
        Odometry parameters:
                left wheel radius: 1
                right wheel radius: 1
                wheel separation: 1
        Publication parameters:
                Publish executed velocity command: 0
                Publication rate: 20
                Publish frame odom on tf: 1
```

Figura 5.26: Ejecución de *ros_control*.

```
friend std::ostream& operator<<(std::ostream& os, const DynamicParams& params)
{
  os << "DynamicParams:\n"
     //
     << "\tOdometry parameters:\n"
     << "\t\tleft wheel radius: "   << params.left_wheel_radius_multiplier  << "\n"
     << "\t\tright wheel radius: "  << params.right_wheel_radius_multiplier << "\n"
     << "\t\twheel separation: "    << params.wheel_separation_multiplier   << "\n"
     //
     << "\tPublication parameters:\n"
     << "\t\tPublish executed velocity command: " << params.publish_cmd << "\n"
     << "\t\tPublication rate: " << params.publish_rate                 << "\n"
     << "\t\tPublish frame odom on tf: " << params.enable_odom_tf;

  return os;
}
```

Figura 5.27: Código del controlador *diff_drive_controller*.

Capítulo 6

Conclusiones y líneas futuras

6.1 Conclusiones

El objetivo de este trabajo ha sido continuar la integración de ROS en la silla de ruedas del departamento de electrónica, iniciada por David Pinedo [11] y Javier León [1] y dotarla de un sistema de navegación.

La navegación se ha implementado mediante el conjunto de paquetes que proporciona *ros_navigation*. Para poder usar este paquete han sido necesarios un conjunto de paquetes complementarios, como *ros_control*, *rviz*, *tf*, *static_transform_publisher*. Además se han usado otros paquetes para hacer el análisis y las pruebas como *rosbag* y *simpleactionclient*.

La parte más compleja del trabajo ha sido la implementación de *ros_control*, ya que la documentación existente es muy reducida. Ha sido necesario realizar un estudio profundo por distintas webs y analizar múltiples proyectos que han servido de ejemplo.También se ha recurrido al soporte en foros de ros [29].

En este trabajo se ha sentado la base de la navegación en la silla de ruedas y gracias a la capacidad modular de ROS, en un futuro será posible añadir nuevos sensores y métodos para corregir los errores de la odometría, así como incluir un sistema de mapeado automático.

6.2 Líneas futuras

En esta sección se exponen las posibles mejoras o trabajos futuros que se pueden realizar en base a este trabajo.

- Implementar un algoritmo que permita corregir los errores producidos por la odometría y realizar una navegación global en un mapa. Esto se puede conseguir con sensores, como un láser o mediante el reconocimiento de etiquetas en el entorno como QR o ARUCO.
- Incorporar un mapeado automático del entorno (SLAM).
- Implementar un reloj común para el *hardware* y el alto nivel para corregir los problemas descritos en el apartado 4.1.6.

Capítulo 7

Pliego de condiciones

A continuación, se detallan los elementos físicos y las herramientas software utilizadas durante el desarrollo de este proyecto.

7.1 Equipos físicos

- Silla de ruedas motorizada
  - Dimensiones 108 cm x 58 cm x 110 cm (largo - ancho - alto)
  - Radio de la rueda 15.5 cm
  - Distancia entre ejes 52.5 cm
  - Reductora x32
  - Batería. Pack de dos baterías de 12V en serie con tecnología AGM.
  - Display LCD serie $i2cS310118$.
  - Teclado matricial 4x3 teclas $S310119$.
  - Joystick.
- Dispositivos añadidos
  - Encoders "HEDS-5500". Dos discos de 500 pulsos: 2000 flancos por vuelta.
  - Sensor de soplido $awm2000v$.
  - Cargador de baterías $EMSPower7969$.
  - Conversor DC-DC $Mascot8862000079$.
  - Controlador de motores $RoboteqAX - 3500$.
  - Microcontrolador $PhilipsLPC2129$.
  - Sensores de ultrasonido $SRF02$.
  - Acelerómetro $ADXL330$.
  - Conversor USB-CAN $LawicelCANUSB$.
- Portátil Lenovo Yoga

7.2 *Software*

- Sistema operativo: Ubuntu 16.04 LTS
- Versión de ROS: Kinetic
- Procesador de textos: Texmaker
- Generador de diagramas: DIA
- Análisis de datos y procesado de los mapas: MATLAB
- Mapas: GIMP y MATLAB

Capítulo 8

Presupuesto

En esta parte se indica el coste total que supone la ejecución del trabajo. En los apartados siguientes aparecen los gastos agrupados según su origen, y en el último apartado se detalla el presupuesto total.

8.1 Recursos *hardware*

Para la realización de este trabajo se han usado los siguientes recursos físicos.

Tabla 8.1: Recursos *hardware*

Concepto	Precio Unit.	Cantidad	Subtotal
Ordenador Lenovo Yoga	700,00 €	1	700,00 €
Silla de Ruedas	1.000,00 €	1	1.000,00 €
TOTAL			1.700,00 €

8.2 Recursos de *software*

Para realizar este trabajo se ha usado software libre, por lo que no tiene ningún costo. El único software privativo que se ha usado es MATLAB para en análisis de los datos en formato *ROS_bag*, ya que es la alternativa que mejor se adapta a las necesidades del proyecto y los rescursos disponibles, ya que se cuenta con la licencia CAMPUS que dispone la UAH.

8.3 Recursos empleados en mano de obra

El trabajo ha sido realizado por un ingeniero. A continuación se desarrollan los costes.

Tabla 8.2: Recursos empleados en mano de obra

Concepto	Precio por hora.	Horas	Subtotal
Ingeniero	50,00 €	500	25.000,00 €
TOTAL			25.000,00 €

8.4 Presupuesto de ejecución material

Tabla 8.3: Presupuesto de ejecución material

Concepto	Precio
Coste recursos hardware	1.700,00 €
Coste recursos software	0,00 €
Coste mano de obra	25.000,00 €
TOTAL	26.700,00 €

8.5 Importe de la ejecución por contrata

A continuación, se incluyen los gastos derivados del uso de las instalaciones donde se ha llevado a cabo el proyecto, cargas fiscales, gastos financieros, tasas administrativas y derivados de las obligaciones de control del proyecto. De igual forma se incluye el beneficio industrial. Para cubrir estos gastos se establece un recargo del 22 % sobre el importe del presupuesto de ejecución material.

Tabla 8.4: Importe de ejecución por contrata

Concepto	Precio
22 % del coste total de ejecución de material	5874,00€

8.6 Honorarios facultativos

Se ha fijado en este proyecto un porcentaje del 7 % sobre el coste total de ejecución por contrata.

Tabla 8.5: Honorarios facultativos

Concepto	Precio
7 % del coste total de ejecución de material	411,18€

8.7 Presupuesto total

Tabla 8.6: Presupuesto total

Concepto	Precio
Presupuesto de ejecución material	26.700,00 €
Importe de la ejecución contratada	5874,00 €
Horarios facultativos	411,18 €
TOTAL (sin IVA)	32985,18 €
IVA (22 %)	7256,73 €
TOTAL	40241,90 €

Bibliografía

[1] J. L. Almazán, "Diseño de módulos software pasa un sistema avanzado robótico de asistencia a la movilidad basada en entorno de desarrollo robótico ros," 2017.

[2] "Página del paquete navigation," http://wiki.ros.org/navigation/Tutorials/RobotSetup.

[3] "Página del paquete simpleactionclient," http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals.

[4] M. M. Romera, "Sistema de posicionamiento absoluto de un robot móvil, función de sensado odométrico y visión de marcas artificiales," 2002.

[5] "How to implement ros_control on a custom robot," https://slaterobots.com/blog/5abd8a1ed4442a651de5cb5b/how-to-implement-ros_control-on-a-custom-robot.

[6] P. M. Peña, "Control y guiado de una silla de ruedas en entornos interiores," 2013.

[7] M. M. Romera, "Navegación autónoma de una silla de ruedas en interiores parcialmente estructurados," 2000.

[8] J. B. Álvarez, "Guiado de una silla de ruedas mediante joystick y soplido por bus can," 2009.

[9] J. B. García, "Guiado semiautomático de una silla de ruedas comandada por un joystick de cabeza a través de bus can," 2009.

[10] A. G. Morcillo, "Beca de investigación sara, universidad de alcalá," 2014.

[11] D. P. Ávila, "Diseño y desarrollo de módulos hardware / software para un sistema avanzado robótico de asistencia a la movilidad (sara)," 2015.

[12] "Paquete urdf," http://wiki.ros.org/urdf.

[13] "Paquete xacro," http://wiki.ros.org/xacro.

[14] "Simulador gazebo," http://wiki.ros.org/gazebo_ros_pkgs.

[15] "Página del paquete move_base," http://wiki.ros.org/move_base.

[16] "Página del paquete costmap2d," http://wiki.ros.org/costmap_2d.

[17] "Página del paquete global_planner," http://wiki.ros.org/global_planner.

[18] K. Zheng, "Ros navigation tuning guide," 2016, pp. 8–9, http://kaiyuzheng.me/documents/navguide.pdf.

[19] "Página del paquete map_server," http://wiki.ros.org/map_server.

[20] "Página del paquete gmapping," http://wiki.ros.org/gmapping.

[21] "Página del paquete fake_localization," http://wiki.ros.org/fake_localization.

[22] "Página del paquete range_sensor_layer," http://wiki.ros.org/range_sensor_layer.

[23] "Página del paquete tf," http://wiki.ros.org/tf.

[24] "Página del paquete tf con aplicacion en navigation," http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF.

[25] "Página del paquete static_transform_publisher," http://wiki.ros.org/static_transform_publisher.

[26] "Página del paquete rviz," http://wiki.ros.org/rviz.

[27] "Página del paquete rviz con aplicacion en navigation," http://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack.

[28] "Página del paquete rosbag," http://wiki.ros.org/rosbag/Tutorials/Recording%20and%20playing%20back%20data.

[29] "Foros de ayuda de ros," https://answers.ros.org/questions/.

Apéndice A

Manual de usuario

A.1 Introducción

En este manual se va a guiar al usuario para que pueda poner en marcha el sistema propuesto en este trabajo.

Todo el software que se va a usar es libre y gratuito, además reciben actualizaciones de forma periódica.

A.2 Instalación del entorno

A.2.1 Introducción

En los siguientes apartados se va a describir los pasos necesarios para instalar el software necesario para poder ejecutar el sistema.

A.2.2 Instalar ROS

Para este trabajo se ha utilizado la versión de ROS *Kinetic* junto con Ubuntu, pero los paquetes son compatibles con versiones posteriores de ROS.

Existe un tutorial de instalación de ROS, solo es necesario seguir todos los pasos que se muestran e instalar la versión completa de ROS (Desktop-Full Install). http://wiki.ros.org/kinetic/Installation/Ubuntu

Hay que tener en cuenta que cada versión de ROS solo es compatible con ciertas versiones de sistema operativo.

A.2.3 Crear el Workspace de *Catkin*

El paquete de ROS Catkin permite compilar nuestros paquetes y poder ser ejecutados en ROS. Catkin viene preinstalado en la versión (Desktop-Full Install) de ROS.

Para poder usar nuestros paquetes es necesario crear un entorno de trabajo, donde colocaremos los paquetes y donde se compilarán.

Listado A.1: Crear entorno de trabajo de Catkin

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

Con estos comandos se ha creado una carpeta en home llamada catkin_ws, donde se encuentra el entorno de trabajo. Los diferentes paquetes creados se copiarán a la carpeta src.

Para poder lanzar los paquetes creados en el entorno de trabajo es necesario añadir las variables de entorno a la consola.

- Se puede añadir de forma temporal con el siguiente comando

Listado A.2: Añadir variables de entorno

```
source devel/setup.bash
```

Pero cada vez que se cierre la consola será necesario volver a instroducirlo.

- Para añadir de forma permanente las variables de entorno es necesario editar el fichero de configuración del terminal de ubuntu:

Listado A.3: Abrir fichero de configuración de la terminal

```
gedit ~/.bashrc
```

Y añadir la siguiente línea al final del fichero de texto

Listado A.4: Línea que hay que añadir

```
source ~/catkin_ws/devel/setup.bash
```

A.2.4 Instalar los paquetes de *ros_control*

Los paquetes de ros_control se descargan mediante la utilidad apt-get. Se ha comprobado que si se compila localmente con catkin_make el rendimiento es mucho peor.

Listado A.5: Instalación de ros_control

```
sudo apt-get install ros-kinetic-ros-control ros-kinetic-ros-controllers
```

A.2.5 Instalar los paquetes de *ros_navigation*

Los paquetes de ros_navigation se descargan mediante la utilidad apt-get. Se ha comprobado que si se compila localmente con catkin_make el rendimiento es mucho peor.

Listado A.6: Instalación de ros_navigationl

```
sudo apt-get install ros-kinetic-navigation
```

Listado A.7: Instalación de la capa de costmap2d range_sensor_layer

```
sudo apt-get install ros-kinetic-range-sensor-layer
```

A.2.6 Compilar los paquetes de SARA

Para poder usar los paquetes creados en este trabajo, primero es necesario compilarlos. Para ello se usará el compilador catkin mencionado en el apartado A.2.3

- Los paquetes proporcionados se copian en el directorio catkin_ws/src.
- Acceder a la raiz del *workspace*

Listado A.8: Acceder al workspace

```
cd catkin_ws
```

- Compilar los paquetes.

Listado A.9: Compilar los paquetes

```
catkin_make
```

A.3 Iniciar el sistema

Para que este trabajo funcione, es necesario lanzar todos los nodos que se han descrito en esta memoria.

Para facilitar el trabajo, cada paquete dispone de uno o varios ficheros *.launch* que permiten lanzar de forma automática todos los nodos de cada uno de los paquetes.

A su vez, los ficheros *.launch* de los paquetes se llaman desde un único fichero *.launch*, de forma que sólo es necesario abrir una terminal y lanzar un comando para iniciar todo el sistema.

En la imagen A.1 se muestra el árbol de ficheros *.launch*.

Para lanzar el trabajo se deben de seguir los siguientes pasos.

- **Activar el modo super usuario**: El paquete canusb necesita permisos de administrador para poder abrir el puerto, por lo tanto es necesario poner el siguiente comando.

Listado A.10: Modo super usuario

```
sudo -s
```

A continuación pedirá la contraseña de administrador, que se debe de introducir.

- **Lanzar el sistema**: Para lanzar el sistema se introduce el siguiente comando en la consola.

Listado A.11: Modo super usuario

```
roslaunch global global_launch.launch
```

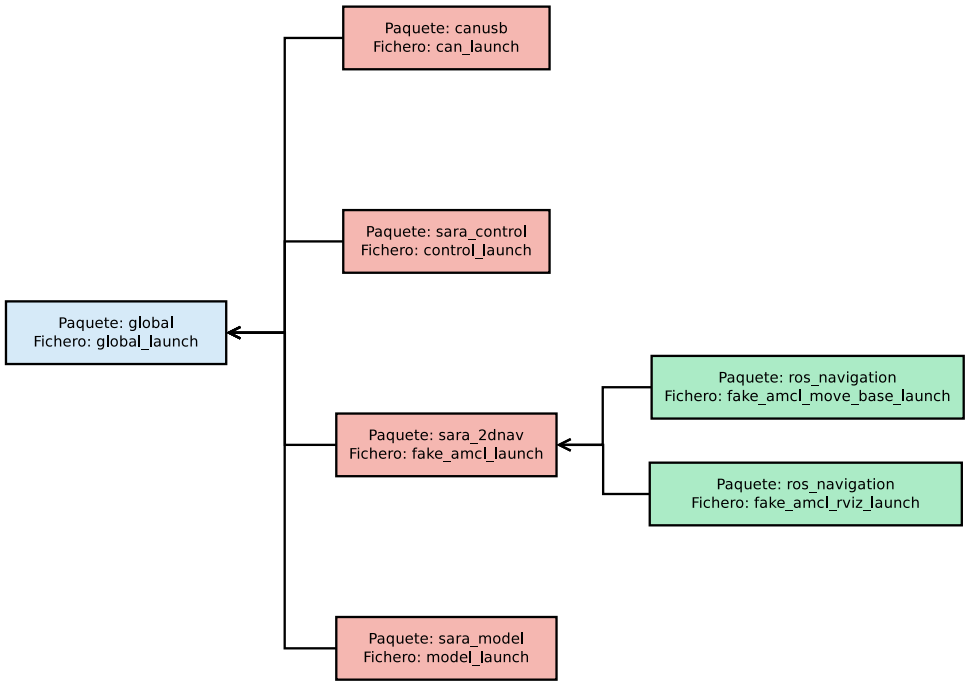


Figura A.1: Diagrama de lanzamiento de los paquetes.

El fichero *global_launch.launch* se encuentra en el paquete *global* y permite lanzar el sistema completo.

En este momento ya tenemos la interface gráfica de rviz lanzada y está lista para aceptar metas. Pero antes es necesario activar el control de las ruedas, pulsando el botón 3 en el teclado numérico. Si esta operación ha sido correcta aparecerá *sincronizado* en la pantalla del teclado numérico. En caso negativo, salir del menú pulsando 0 y volver a sincronizar pulsando 3.

A.4 Cambiar mapa y posición inicial

Ambas cosas se cambian en el fichero fake_amcl_launch.

Los mapas se encuentran en la carpeta maps del paquete sara_2dnav. El nombre del mapa se debe introducir en la siguiente línea.

Listado A.12: Nombre del mapa

```
<arg name="map" default="uah_map.yaml" />
```

La posición inicial se fija en los parámetros de fake_localization:

Listado A.13: Posición inicial

```
<param name="delta_x" value="-3.4" />
<param name="delta_y" value="-6.5" />
<param name="delta_yaw" value="0.3" />
```

Recordar que el parámetro *delta_yaw* no funciona correctamente a fecha de entrega de este TFG, como se explica en el apartado 5.8.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá